

目 录

第 1 章 概述	1
1.1 回顾测试的发展	1
1.2 什么是软件测试	3
1.2.1 IEEE 的定义	3
1.2.2 测试在软件开发中的角色	4
1.3 为什么要进行软件测试	4
1.4 测试的目的	6
1.4.1 证明	6
1.4.2 检测	6
1.4.3 预防	6
1.5 业界的软件测试现状	7
1.6 软件测试中的误区	11
1.7 本章小结	12
第 2 章 白盒测试和黑盒测试	13
2.1 白盒测试	13
2.1.1 什么是白盒测试	13
2.1.2 为什么要进行白盒测试	14
2.1.3 白盒测试的常用技术	14
2.1.4 一个白盒测试的例子	16
2.2 黑盒测试	18
2.2.1 什么是黑盒测试	18
2.2.2 为什么要进行黑盒测试	19
2.2.3 黑盒测试的常用技术	19
2.2.4 一个黑盒测试的例子	20
2.3 白盒测试和黑盒测试的比较	21
2.3.1 白盒测试的优缺点	22
2.3.2 黑盒测试的优缺点	22
2.3.3 灰盒测试	23
2.4 本章小结	23
第 3 章 测试覆盖率	24
3.1 覆盖率概念	24
3.2 常见的逻辑覆盖	24

3.2.1	语句覆盖	24
3.2.2	判定覆盖	25
3.2.3	条件覆盖	26
3.2.4	判定条件覆盖	28
3.2.5	路径覆盖	29
3.2.6	逻辑覆盖小结	30
3.3	功能覆盖率	30
3.4	面向对象的覆盖率	31
3.4.1	继承上下文覆盖	32
3.4.2	基于状态的上下文覆盖	33
3.4.3	基于线程的上下文覆盖	35
3.5	其他覆盖率	36
3.5.1	函数覆盖	36
3.5.2	指令块覆盖	36
3.5.3	判定路径覆盖	37
3.5.4	更改条件判定覆盖	37
3.5.5	分支条件组合覆盖	39
3.5.6	过程到过程路径覆盖	39
3.5.7	Z 路径覆盖	39
3.5.8	ESTCA 覆盖	41
3.5.9	LCSAJ 覆盖	42
3.6	如何使用覆盖率	43
3.6.1	基本原则	43
3.6.2	一个选择建议	44
3.6.3	使用最少测试用例来达到覆盖	45
3.7	本章小结	47
第4章	程序分析技术	48
4.1	文本视角	48
4.2	句法视角	49
4.3	控制流视角	54
4.4	数据流视角	55
4.5	计算流视角	58
4.5.1	故障插入	59
4.5.2	变体分析	59
4.5.3	敏感性分析	62
4.6	功能视角	63
4.7	本章小结	64

第5章 测试分析技术	65
5.1 面向规格的测试	65
5.1.1 测试独立于规格技术	66
5.1.2 测试依赖于规格技术	68
5.2 面向实现的测试	75
5.2.1 面向结构的测试	76
5.2.2 面向影响的测试	77
5.2.3 面向传递的测试	79
5.3 面向错误的测试	86
5.3.1 基于错误的测试	87
5.3.2 基于故障的测试	88
5.3.3 基于风险的测试	88
5.3.4 可能的正确性	89
5.4 混合测试技术	89
5.5 本章小结	90
第6章 单元测试	91
6.1 什么是单元测试	91
6.1.1 单元测试的目的	92
6.1.2 单元测试和集成测试的区别	92
6.1.3 单元测试和系统测试的区别	93
6.2 为什么要进行单元测试	93
6.3 单元测试环境	95
6.4 单元测试策略	97
6.4.1 【策略一】由顶向下的单元测试策略	97
6.4.2 【策略二】由低到上的单元测试策略	97
6.4.3 【策略三】孤立测试	98
6.4.4 综合测试	98
6.5 单元测试分析	99
6.5.1 模块接口	99
6.5.2 局部数据结构	99
6.5.3 独立路径	100
6.5.4 出错处理	100
6.5.5 边界条件	100
6.6 单元测试用例设计思路	101
6.6.1 为系统运行设计用例	101
6.6.2 为正向测试设计用例	101
6.6.3 为逆向测试设计用例	101
6.6.4 为满足特殊需求设计用例	102

6.6.5	为代码覆盖设计用例	102
6.6.6	为覆盖率指标完成设计用例	102
6.7	单元测试过程	103
6.7.1	完成测试计划	104
6.7.2	获取测试集	106
6.7.3	度量测试单元	107
6.8	单元测试工具介绍	110
6.9	单元测试应坚持的原则	110
6.10	我们的问题	111
6.11	本章小结	112
第7章	集成测试	113
7.1	什么是集成测试	113
7.1.1	集成测试与系统测试的区别	113
7.1.2	集成测试关注的重点	114
7.1.3	集成测试和开发的关系	114
7.1.4	集成测试的层次	115
7.2	集成测试策略	115
7.2.1	大爆炸集成	116
7.2.2	自顶向下的集成	117
7.2.3	自底向上的集成	119
7.2.4	三明治集成	121
7.2.5	修改过的三明治集成	122
7.2.6	基于集成	123
7.2.7	分层集成	124
7.2.8	基于功能的集成	126
7.2.9	高频集成	127
7.2.10	基于进度的集成	129
7.2.11	基于风险的集成	130
7.2.12	基于事件(消息)的集成	130
7.2.13	基于使用的集成	131
7.2.14	客户/服务器的集成	132
7.2.15	分布式集成	132
7.3	集成测试分析	133
7.3.1	体系结构分析	133
7.3.2	模块分析	134
7.3.3	接口分析	136
7.3.4	风险分析	137
7.3.5	可测试性分析	138

7.3.6	集成测试策略分析	138
7.3.7	常见的集成测试故障	138
7.4	集成测试用例设计思路	139
7.4.1	为系统运行设计用例	139
7.4.2	为正向测试设计用例	140
7.4.3	为逆向测试设计用例	140
7.4.4	为满足特殊需求设计用例	140
7.4.5	为高覆盖设计用例	141
7.4.6	测试用例补充	141
7.4.7	注意事项	141
7.5	集成测试过程	141
7.5.1	计划阶段	141
7.5.2	设计阶段	142
7.5.3	实现阶段	143
7.5.4	执行阶段	144
7.6	集成测试环境	145
7.7	集成测试工具介绍	146
7.8	集成测试应坚持的原则	146
7.9	本章小结	147
第8章	系统测试	148
8.1	系统测试概念	148
8.2	系统测试方法	149
8.2.1	功能测试	149
8.2.2	协议一致性测试	150
8.2.3	性能测试	152
8.2.4	压力测试	154
8.2.5	容量测试	156
8.2.6	安全性测试	156
8.2.7	恢复性测试	159
8.2.8	备份测试	160
8.2.9	GUI 测试	160
8.2.10	健壮性测试	163
8.2.11	兼容性测试	166
8.2.12	可用性测试	167
8.2.13	可安装性测试	168
8.2.14	文档测试	171
8.2.15	在线帮助测试	172
8.2.16	数据转换测试	172

8.3	系统测试过程	173
8.3.1	完成系统测试计划	175
8.3.2	完成系统测试用例	178
8.3.3	评审/审批系统测试计划	178
8.3.4	执行系统测试	179
8.4	本章小结	181
第9章	可靠性与可靠性测试	182
9.1	基本概念	182
9.1.1	什么是软件可靠性	182
9.1.2	错误、缺陷、故障和失效	183
9.1.3	软件可靠性指标	184
9.1.4	软件和硬件可靠性区别	186
9.2	可靠性指标分配	187
9.3	可靠性预计	188
9.3.1	计数法	188
9.3.2	应力法	188
9.4	可靠性分析方法	189
9.4.1	FMEA	189
9.4.2	CA	191
9.4.3	FTA	192
9.4.4	ETA	193
9.4.5	SCA	193
9.5	软件可靠性测试	194
9.5.1	可靠性测试流程	195
9.5.2	可靠性模型介绍	198
9.5.3	一个可靠性数据分析例子	202
9.6	软件可靠性工程	205
9.7	可靠性标准和可靠性工具	205
9.7.1	可靠性标准	205
9.7.2	可靠性工具	207
9.8	本章小结	208
第10章	其他专项性测试	210
10.1	可接受性测试	210
10.2	Alpha 测试	211
10.3	Beta 测试	212
10.4	标杆测试	213
10.5	配置测试	214

10.6	外场测试	214
10.7	SQL 测试	215
10.8	2000 年测试	215
10.9	回归测试	216
10.10	本章小结	218
第 11 章	软件质量透视	219
11.1	质量的定义	219
11.2	质量的预防和检测	220
11.3	如何提高软件产品的质量	221
11.3.1	流程对质量的贡献	221
11.3.2	流程与技术	223
11.3.3	全面质量管理	224
11.3.4	关注测试	233
11.3.5	组织、流程和人	234
11.4	质量标准	324
11.5	本章小结	237
第 12 章	软件验证和确认	238
12.1	基本概念	238
12.2	软件验证和确认计划	240
12.2.1	SVVP 步骤	241
12.2.2	SVVP 的 7 个主题	242
12.3	验证和确认任务分析	248
12.3.1	关键性分析	248
12.3.2	可跟踪性分析	250
12.3.3	评估	251
12.3.4	接口分析	252
12.3.5	测试	253
12.4	生命周期各阶段活动	253
12.4.1	管理阶段的验证和确认	253
12.4.2	概念阶段的验证和确认	255
12.4.3	需求阶段的验证和确认	255
12.4.4	设计阶段的验证和确认	256
12.4.5	实现阶段的验证和确认	257
12.4.6	测试阶段的验证和确认	258
12.4.7	安装和校验阶段的验证和确认	258
12.4.8	运行和维护阶段的验证和确认	260
12.4.9	验证和确认任务总结	261

12.5	验证和确认的报告	274
12.5.1	标准要求的报告	275
12.5.2	标准可选报告	275
12.6	本章小结	275
第13章	软件质量保证	277
13.1	基本概念	277
13.1.1	目标	278
13.1.2	执行的承诺	278
13.1.3	执行的能力	278
13.1.4	执行的活动	279
13.1.5	度量分析	281
13.1.6	验证实现	281
13.2	SQA 实施过程	281
13.2.1	建立 SQA 组织	282
13.2.2	选择 SQA 任务	283
13.2.3	产生/维护 SQA 计划	297
13.2.4	实施 SQA 计划	299
13.2.5	产生/维护 SQA 规程	299
13.2.6	标识 SQA 培训	299
13.2.7	标识/选择 SQA 工具	300
13.2.8	改进项目 SQA 过程	300
13.3	本章小结	301
第14章	需求测试	302
14.1	需求测试概述	302
14.1.1	什么是需求	302
14.1.2	测试需求	307
14.2	通过评审来测试需求	307
14.2.1	需求评审中的常见风险	307
14.2.2	需求评审检查表	308
14.3	通过用例设计来测试需求	313
14.4	需求建模测试	316
14.4.1	统一建模语言	317
14.4.2	消息顺序图(MSC)	320
14.4.3	分析建模工具介绍	321
14.4.4	需求的形式化描述	324
14.5	基于原型的测试	325
14.5.1	原型的目的	325

14.5.2	原型的种类	325
14.5.3	原型的测试方法	326
14.6	本章小结	327
第15章	设计测试	328
15.1	设计测试概述	328
15.1.1	什么是设计	328
15.1.2	软件构架设计	330
15.1.3	概要设计和详细设计	334
15.2	设计的评审	336
15.2.1	设计查检表	336
15.2.2	构架设计评审方法	340
15.2.3	软件构架评价最佳工业实践	345
15.3	SDL 及相关测试	352
15.3.1	SDL 介绍	352
15.3.2	SDL 基本概念	353
15.3.3	SDL 结构	356
15.3.4	SDL 测试	358
15.4	本章小结	365
第16章	同行评审	366
16.1	基本概念	366
16.2	同行评审的一般过程	367
16.2.1	计划阶段	367
16.2.2	实施被选择的同行评审过程	370
16.2.3	同行评审过程度量	370
16.2.4	同行评审的评审/审计	370
16.3	走读	371
16.3.1	过程目标	371
16.3.2	特定的角色和职责	371
16.3.3	输入	371
16.3.4	入口标准	371
16.3.5	过程	371
16.3.6	出口标准	372
16.3.7	输出	372
16.4	技术评审	372
16.4.1	过程目标	373
16.4.2	特定的角色和职责	373
16.4.3	输入	373

16.4.4	入口标准	373
16.4.5	过程	373
16.4.6	出口标准	374
16.4.7	输出	374
16.5	正规检视	374
16.5.1	正规检视小组	375
16.5.2	正规检视过程	379
16.5.3	正规检视常用表格	387
16.6	本章小结	393
第 17 章	测试经验总结	395
17.1	软件测试的 10 大原则	395
17.1.1	原则 1: 测试是一个持续进行的过程, 而不是一个阶段	395
17.1.2	原则 2: 测试必须被计划、被控制, 并且被提供时间和资源	395
17.1.3	原则 3: 测试应当分级别	396
17.1.4	原则 4: 测试应当有重点	397
17.1.5	原则 5: 测试不是为了证明程序的正确性, 而是为了证明 程序不能工作	397
17.1.6	原则 6: 测试是不可能穷尽的, 当测试出口条件满足时就 可以停止测试	398
17.1.7	原则 7: 测试是开发的朋友, 不是开发的敌人	398
17.1.8	原则 8: 测试人员应公正地测试, 如实地记录和报告缺陷	399
17.1.9	原则 9: 测试自动化能解决一部分问题, 但不是全部	399
17.1.10	原则 10: 测试不能仅仅包括功能性的验证, 还应当包含 性能、可靠性、可维护性、安全性等方面的验证	399
17.2	软件测试的 10 个最佳实践	400
17.2.1	实践 1: 尽早地、频繁地进行测试是降低项目成本, 提高 质量的一个好方法	400
17.2.2	实践 2: 尽早产生一个综合的主测试计划	400
17.2.3	实践 3: 对质量要求较高或大型复杂的产品成立独立 的测试组	401
17.2.4	实践 4: 在每个开发阶段, 使用测试和评价的结果作为是否 可以通过的标准	401
17.2.5	实践 5: 开发和维护一个测试需求和目标的风险优先级列表	401
17.2.6	实践 6: 把测试件作为产品的一部分等同管理, 使用相同的 评价标准和过程	402
17.2.7	实践 7: 提供集成化的测试工具和测试基础支持	402
17.2.8	实践 8: 加强测试度量工作和缺陷分析工作, 不断地 改进测试	404

17.2.9 实践 9：加强测试的培训并且为测试人员提供技能发展的通道	405
17.2.10 实践 10：加强沟通和交流，让项目组内所有人员都了解测试的工作及其重要性	405
17.3 本章小结	406
附录 A 常见测试术语	407
附录 B 测试技术分类	427
附录 C 常见的编码错误	430
附录 D 经典测试网站	433
附录 E 参考资料	437

第1章 概述

在学习本章时，可以从以下几个问题进行考虑：

1. 什么是软件测试？
2. 软件测试是怎么发展起来的？
3. 为什么要进行软件测试？
4. 软件测试的目的是什么？
5. 常见的软件测试误区有哪些？

1.1 回顾测试的发展

测试是一个极其宽广的概念，并且涉及到我们生活的各个方面。一辆汽车从流水线上下来需要经过各种测试之后才能最终到达用户手中。作为用户，他们希望所使用的产品是可靠的、符合需要的。那么，怎样才算是可靠的、符合需要的呢？最简单的方法是通过建立一些异常的环境，看产品在这些环境下能否正常工作。这就是测试。

可以说，测试是随着社会化生产应运而生的，不是近代才出现的产物。但是，我们这里强调的是对软件产品的测试。软件测试是软件工程的一个范畴，或者说是软件工程的一个部分。这主要是因为测试活动是一个工程性的活动而不是简单、孤立的活动。

软件测试最早可以追溯到软件开发的初期。随着计算机的诞生，就开始了软件开发和软件测试。由于早期的计算机运行性能比较差，软件的可编程性范围也比较狭窄，错误主要集中在元器件的不稳定上。在这一阶段还没有系统意义上的软件测试，更多的是一种调试性测试。测试用例的设计和选取是在随机的基础上，凭借测试人员一定的经验进行的。测试更多的是为了证明系统可以运行起来。

20世纪50年代后期到20世纪60年代，高级语言相继诞生并得到了广泛的应用，测试的重点逐渐转入到高级语言编写的系统中来。与以往不同的是程序的复杂性增强了。但是，由于受到硬件系统的制约，软件相对而言仅占系统的次要位置。软件正确性的把握主要依赖编程人员的水平。因此，测试理论和方法在这一阶段的发展比较缓慢。

20世纪70年代以后，随着计算机处理速度的飞速提高，内存和外存（主要是硬盘）容量的快速增加，软件在整个系统中的重要性变得越来越高，软件的规模越来越大，可视化编程环境、日益完善的软件分析设计方法（如面向对象的分析设计概念的产生）以及新的软件开发过程模型的提出（如螺旋模型，增量模型等）使得大型软件的开发成为可能；同时，由于软件规模和复杂度的急剧增加，软件的可靠性面临着危机；软件测试在这一阶段承受了挑战。许多测试理论和测试方法相继诞生，逐渐形成一套体系。同时，这一阶段也孕育、培养出了一批出色的测试专家^[1]。

随着软件产业化发展，人们对软件的质量、成本和进度提出了较高的要求。质量的

控制已经不再是传统意义上的软件测试。传统的软件测试是基于代码运行的，只有在软件开发的后期才能介入。然而，产业界（如 TRW、Nippon Electric 和 Mitre Corp 以及其他一些公司）的大量研究表明，设计活动引入的错误占软件过程中出现所有错误（和最终的缺陷）数量的 50%~65%。根据 IBM 的研究结果，假定在分析阶段发现的错误其改正成本为 1 个货币单位，那么在测试之前（设计编码阶段）发现一个错误的修改成本约为 6.5 个货币单位，在测试时（集成测试，系统测试和验收测试）发现一个错误的修改成本约为 15 个货币单位，而在发布之后（已经交到用户手上）发现一个错误的修改成本约为 60~100 个货币单位。该比例同样也适用于发现一个错误需要的时间代价。从图 1-1 的两条曲线可看出这个比例。

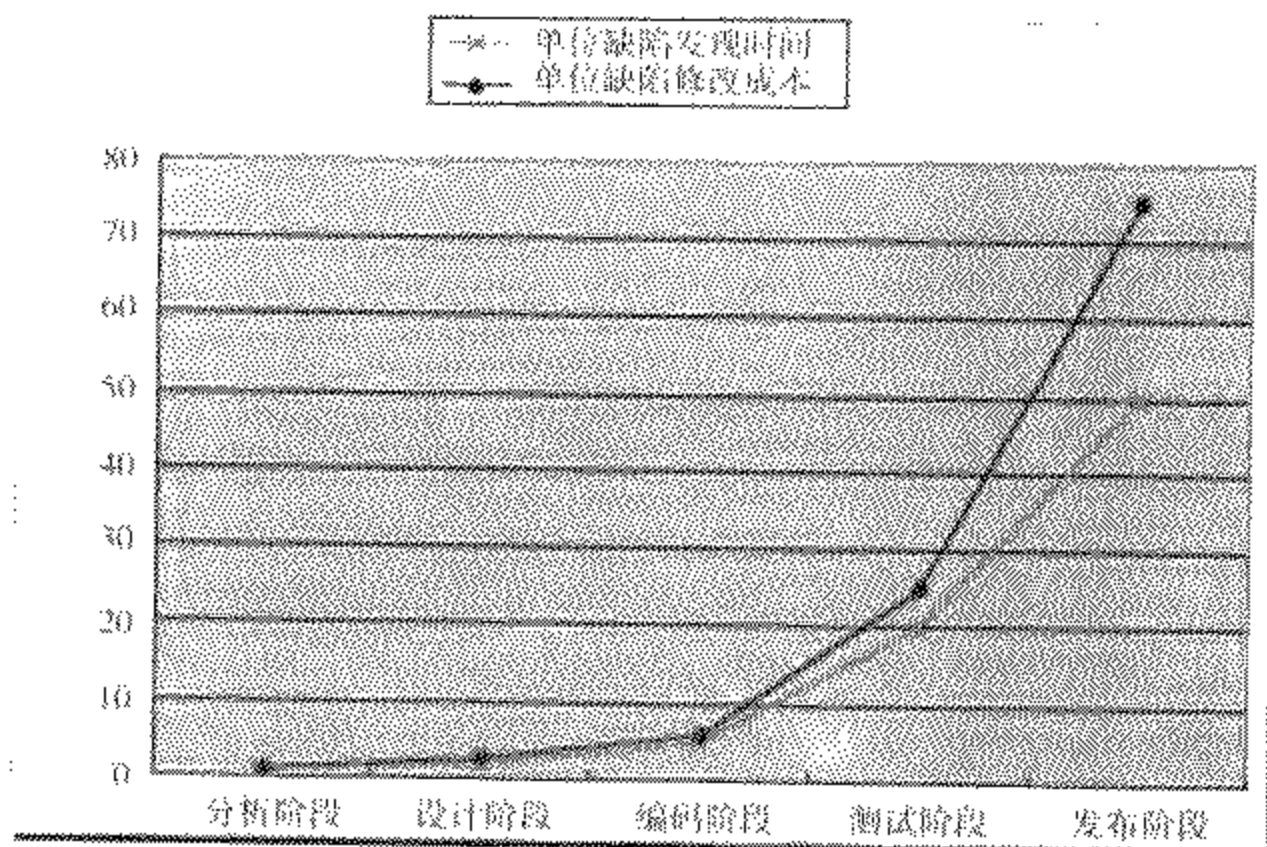


图 1-1 缺陷修改成本趋势图

IBM 的研究结果还表明：缺陷存在放大趋势^[6]。如果在需求阶段漏过一个错误，该错误可能会引起 n 个设计错误。 n 称为放大系数。一般而言，不同阶段其 n 不同。经验表明，从概要设计到详细设计的错误放大系数大约为 1.5，从详细设计到编码阶段的错误放大系数大约为 3。图 1-2 表示了缺陷放大模型大致状况。

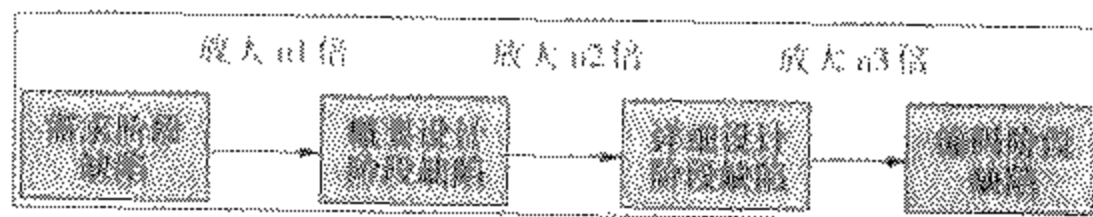


图 1-2 缺陷放大模型图

因为有上面这些内在因素的制约，就不难想象为什么很多软件产品在其开发过程中投入了大量的时间和金钱在没完没了的系统测试上，而最后得到的产品却依旧是低质量的软件。在经过了大量的失败实践之后，人们逐渐发现，一个软件的成功不可能仅仅依赖于软件开发技术是否先进。相反，软件产业化要求有一个规范的软件开发过程，一个全局的质量控制体系。一个好的软件开发过程为软件的开发指明了一条通向成功的捷径。在此要求

下,许多优秀的软件开发过程,开发规范应运而生,如CMM、IPD、RUP等。在这些过程中,测试已经不再是一个编码后才进行的活动,而是一个基于软件开发整个生命周期的质量控制活动。测试的概念也扩展到了静态测试范畴。

软件测试的V模型说明了何时应进行测试。但是,该模型仅涉及到单元测试、集成测试、系统测试和验收测试的过程,以及这些测试与前期阶段的对应关系。其实,在开发过程中,测试始终在扮演着验证和确认的角色。如在功能性需求分析阶段,除了要考虑系统测试范围的内容外,还必须包括功能需求本身的测试。从需求考虑系统测试范围,主要是考虑系统测试应当按照功能需求的内容测试系统是否符合需求;而需求本身的测试则是测试需求的完整性、一致性、无冲突性等。软件测试V模型如图1-3所示。

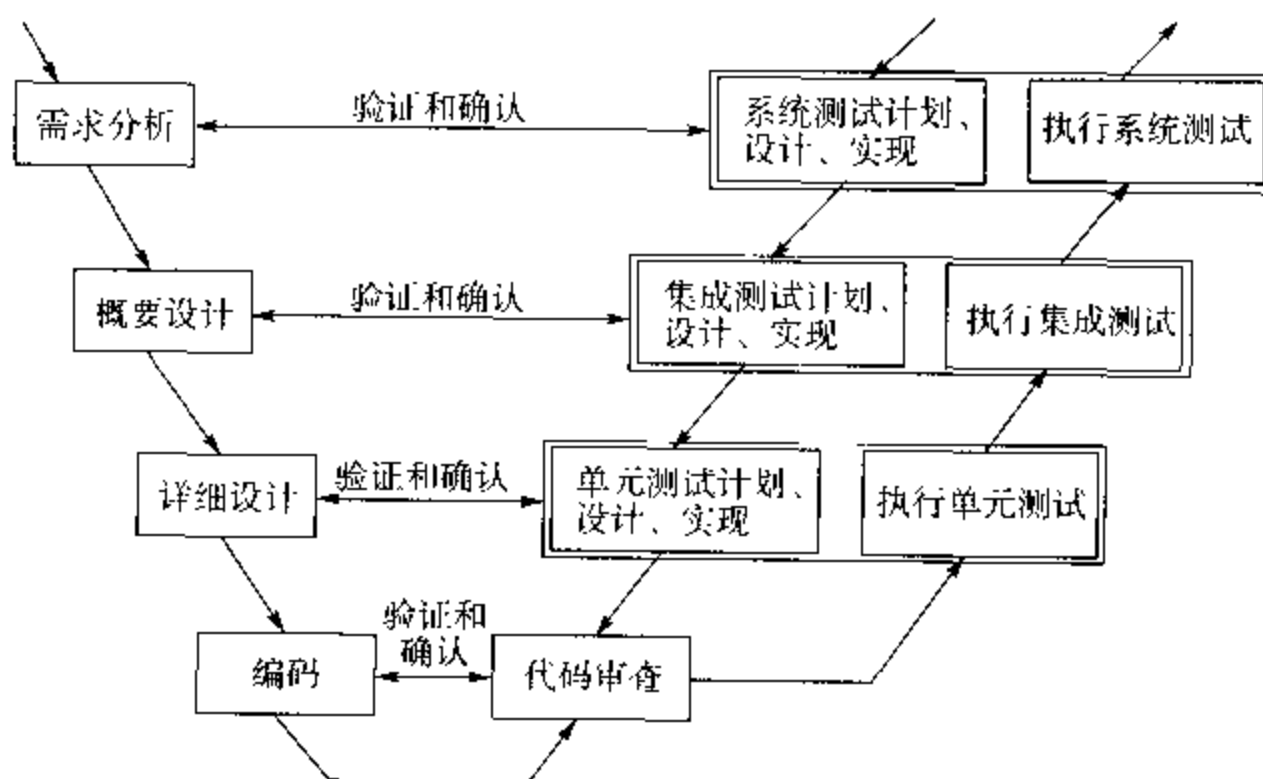


图 1-3 测试 V 模型

1.2 什么是软件测试

简单地说,如果你写了一段代码,我来帮你查看代码并找出里面的错误,这就是测试。曾经有位大师说过:软件测试目的在于发现错误;一个好的测试用例在于发现从前未发现的错误;一个成功的测试是发现了从前未发现的错误的测试^[2]。

1.2.1 IEEE 的定义

1983年,IEEE提出了软件工程标准术语,软件测试定义为:

“使用人工和自动手段来运行或测试某个系统的过程,其目的在于检验它是否满足规定的功能或是弄清预期结果与实际结果之间的差别。”

该定义明确提出了软件测试以检验是否满足需求为目标。

1.2.2 测试在软件开发中的角色

为了更好地理解测试，必须了解测试在软件开发中所扮演的角色。主要如下^[3]：

- 测试是执行或者模拟一个系统或者程序的操作。
- 测试是为了建立一个信心，即软件是按照它所要求的方式执行的，而不会执行它不被希望的操作。
- 测试是带着发现问题和错误的意图来分析程序的。
- 测试是度量程序的功能和质量的。
- 测试是评价程序和项目工作产品的属性和能力的，并且评估其是否获得了期望和可接受的结果。
- 测试除了包括执行代码的测试，还包括检视和结构化同行评审。

测试包含了上面所有内容。

1.3 为什么要进行软件测试

AirLie 软件咨询中心提出了 5 个主要的原因^[3]，它们是：

- 一个糟糕的测试程序可能导致任务的失败，更严重的是可能影响操作的性能和可靠性，并且可能会导致在维护阶段花费巨大的成本，这一点从 1.1 节 IBM 的调查中可以得到。
- 一个好的测试程序是项目的主要成本。复杂的项目需要花费超过项目一半以上的成本在软件测试和验证上。为了使测试有效，必须提前花费适当的时间在计划和组织测试上面。
- 一个好的程序可以极大地帮助你定义需求和设计。这有助于项目在一开始时就步入正轨，并且它对整个项目的成功有着重要的影响。
- 一个好的测试可以迫使你在工作时必须去面对和处理问题，并且会使重新工作或修改缺陷的成本变得很低。
- 一个好的测试不能弥补一个糟糕的软件项目，但是的确有助于发现许多问题并且至少使得你尽早知道你处在问题当中。

只要是人，都会犯错。即使是一个很优秀的程序员，也会犯低级性的错误。正因如此，测试是必须的。常见导致错误的根源有：

- 缺乏有效的沟通，或者没有进行沟通

现在的软件开发已经不是一个人的事情了，往往涉及到多个人，甚至几十、几百个人。同时软件的开发还需要与不同的人，不同的部门进行沟通。如果在沟通方面表现不力，最后会导致产品无法集成，或者集成出来的产品无法满足用户需求。

- 软件复杂度

软件越复杂就越容易出错。在当今的软件开发中，对于一些没有经验的人来说，软件复杂性可能是难以理解的。图形化界面、客户/服务器和分布式的应用、数据通信、大规

模的关系数据库、应用程序的规模等指数般地增加了软件的复杂度。面向对象技术也有可能增加软件复杂度，除非能够被很好地工程化。

- 编程错误

编程错误是程序员经常会犯的错误，包括语法错误、语义错误、拼写错误、编程规范错误。有很多错误可以通过编译器直接找到，但是遗留下来的错误就必须通过严格的测试才能发现。

- 不断变更的需求

在实际项目开发过程中，不断变更的需求是项目失败的最大杀手。用户可能不知道变更的影响，或者知道影响却还是需要进行变更。这些会引起项目重新设计，工程的重新安排，对其他项目产生影响。已完成的工作可能不得不重做或推翻。硬件需求可能也会受到影响。如果存在许多小的变更或者任何大的改动，由于项目中不同部分间可知和不可知的依赖关系，这样就会产生问题，跟踪变更的复杂性也可能引入错误。项目开发人员的积极性也会受到打击。在一些快速变化的商业环境下，不断变更的需求可能是一种残酷的现实。在此情况下，管理人员必须了解结果的风险，QA 工程师和测试工程师必须适应和计划进行大规模的测试来防止不可避免的 Bug 出现无法控制的局面。

- 时间的压力

进度压力是每个从事软件开发人员都会碰到的问题。为了抢占市场，我们必须比竞争对手早一步把产品提供出来，于是不合理的进度安排就产生了。不断加班加点，最终导致大量错误的产生。另一方面，由于软件项目的时间安排是最难的，通常需要很多猜测性的工作。因此，当最后期限来临时，错误也就伴随发生了。

- 缺乏文档的代码

由于人员的变动和产品的生命周期演进，在一个组织中很难保证一个人一直待在某个产品当中。因此对于后面进入产品的人员来说，去读懂和维护一个没有文档的糟糕代码是一个灾难。最终的结果只会导致更多的问题。

- 软件开发工具

当产品开发依赖于某些工具时，那么这些工具本身隐藏的问题可能会导致产品的缺陷。因此，在选择软件开发工具时，尽可能选择比较成熟的产品，不要去追求技术最新的开发工具。这类工具往往本身还存在很多问题。

- 人员的自大

我们经常会发现人们普遍喜欢说：

“没问题”

“很简单”

“我可以在几小时内解决那个问题”

“修改那些老代码应当是很简单的”

而不是说：

“那会增加很多复杂性，可能会导致很多错误”

“如果我们要做那个的话，我们将无能为力”

“我无法估计可能要多长时间，除非我能进一步进行观察和研究”

“我们无法搞清楚那些混乱的代码到底在做什么事情”

如果存在太多的“没问题”，问题也就产生了。

1.4 测试的目的

从历史的观点来看，测试关注于执行软件来获得软件在可用性方面的信心并且证明软件能够满意地工作。这引导测试把重点投入在检测和排除缺陷上。现代的软件测试持续了这个观点。同时，还认识到许多重要的缺陷主要来自于对需求和设计的误解、遗漏和不正确。因此，早期的结构化同行评审被用于帮助预防编码前的缺陷。证明、检测和预防已经成为一个良好的测试的重要目标。这一发展过程如图 1-4 所示。

证明	检测	预测
表明软件能够工作	发现错误	管理质量
20 世纪 60 年代	20 世纪 70 年代中期	20 世纪 90 年代

图 1-4 测试目的的演进

1.4.1 证明

- 获取系统在可接受风险范围内可用的信心；
- 尝试在非正常情况和条件下的功能和特性；
- 保证一个工作产品是完整的并且可用或者可被集成。

1.4.2 检测

- 发现缺陷、错误和系统不足；
- 定义系统的能力和局限性；
- 提供组件、工作产品和系统的质量信息。

1.4.3 预防

- 澄清系统的规格和性能；
- 提供预防或减少可能制造错误的信息；
- 在过程中尽早检测错误；
- 确认问题和风险，并且提前确认解决这些问题和风险的途径。

1.5 业界的软件测试现状

QAI (Quality Assurance Institute) 每年在其软件测试国际会议上都会进行一次软件测试调查。这个调查给测试人员提供了软件测试的当前状态。许多调查结果每年都没有太大的变化。下面是一些调查结果。

询问参与被调查人员所在组织是否有文档化了的和维护的测试标准/过程。其中 71% 的人回答是有，见图 1-5。在回答否的人中，大约有三分之二的人回答他们的组织正在把这项工作纳入他们的计划中。

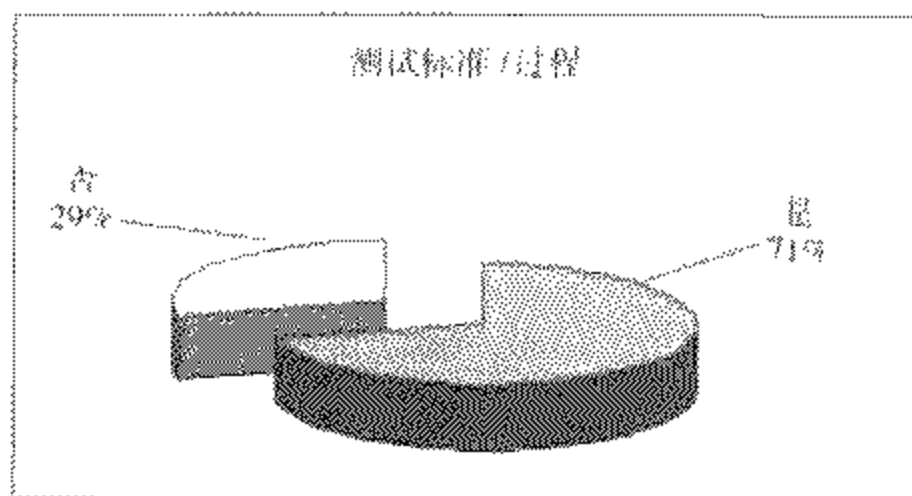


图 1-5 测试标准/过程调查结果

询问参与被调查人员所在组织是否计划在下一年度，或下两个年度进行软件测试过程的改进。88% 的人员回答是肯定的。在这些肯定答复中，建立度量 and 过程改进已经成为计划改进措施中最优先的部分。其他改进计划包括：风险管理，配置管理和培训，如图 1-6 所示。

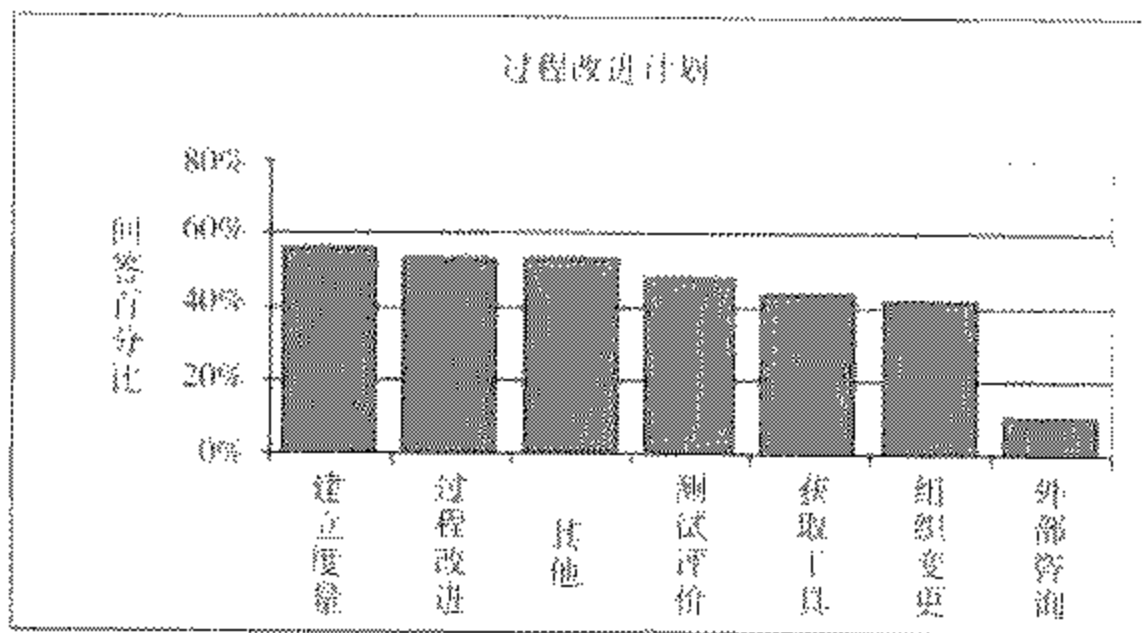


图 1-6 过程改进计划调查结果

询问参与被调查人员所在组织一般是在什么硬件平台上进行软件测试的情况是，从 1995 年以后，客户/服务器已经成为了主要的平台。在前几年调查中，PC 机是一个重要的平台。其他包括 AS 400，Unix 和 Tandem，见图 1-7。

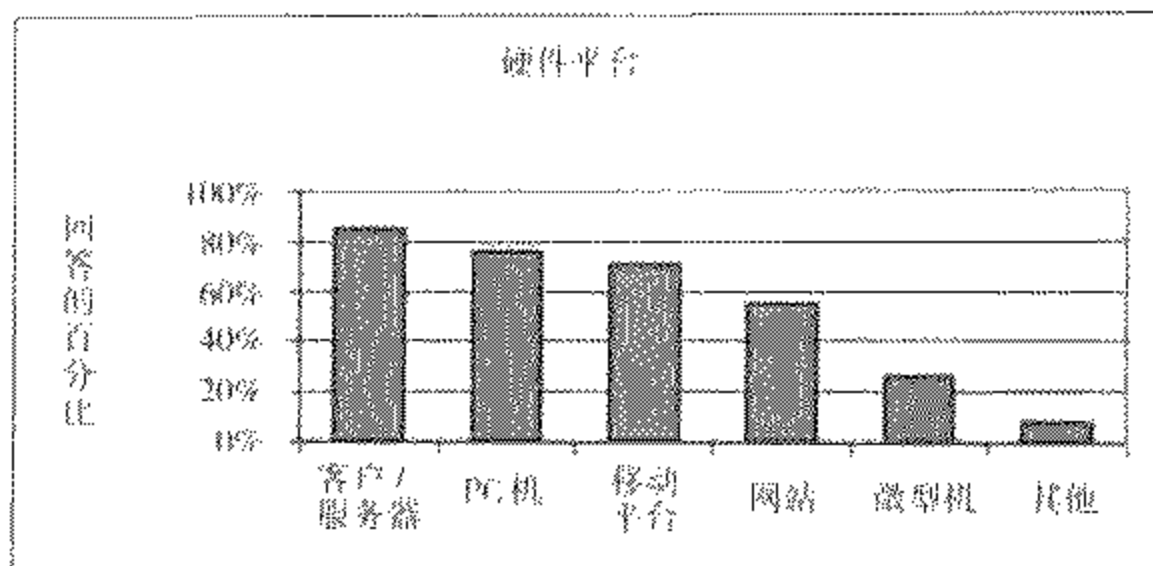


图 1-7 测试硬件平台调查结果

询问参与被调查人员“你们特定测试活动的最佳描述是什么？”41%的人回答是项目测试组，28%的人回答是独立测试组，21%是其他，12%是最终用户、2%是程序员。其他包括管理和审查活动。2001 年的调查中，项目测试组超过了独立测试组 12%，见图 1-8。

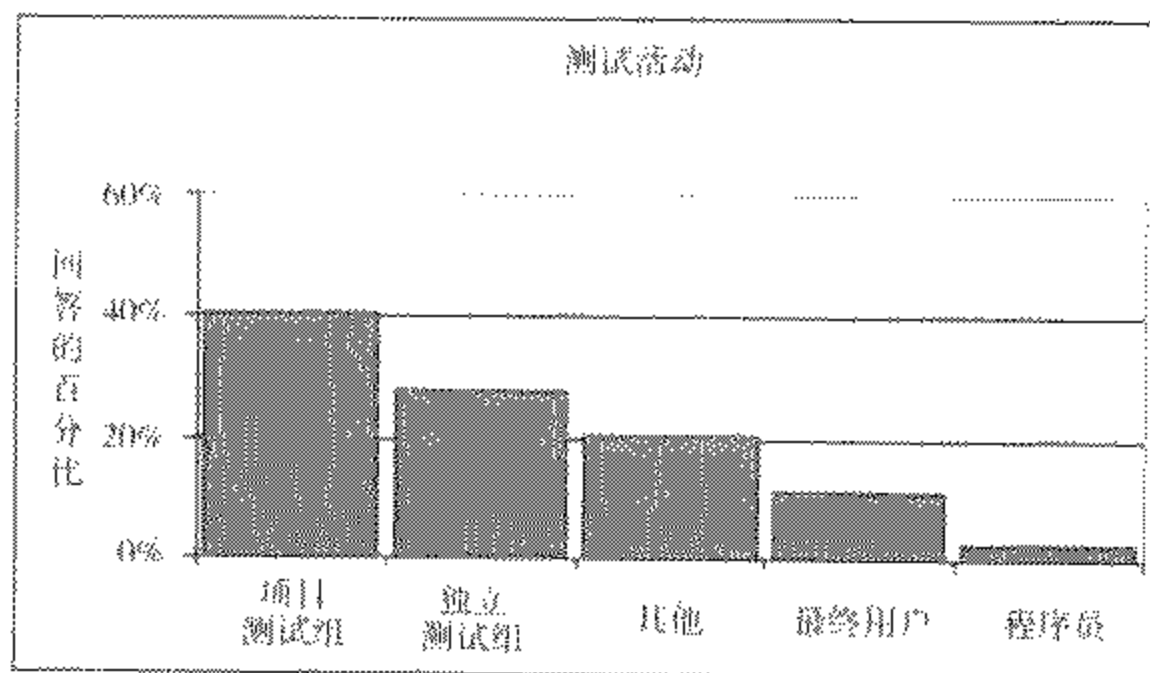


图 1-8 测试活动调查结果

询问参与被调查人员“你们的组织是否对测试人员进行培训？”63%的人员回答是肯定的。培训的类型包括 QAI 讨论会、内部的正式课程、工具供应商的培训和手把手的实际工作指导。

询问参与被调查人员“什么样的测试产品被开发出来用于你们的测试过程？”超过 80% 的人员回答他们使用详细的测试计划、测试用例、测试环境和测试数据。当问到“测试产品被维护和重用的程度是多少？”时，33% 的人回答一直是这么做的，59% 的人员回答有时是这么做的，8% 的人员从来没有这么做过。见图 1-9。

询问参与被调查人员“你们的组织进行什么样的静态测试？”61% 的人员回答是终端用户/客户评审。其他包括 56% 的项目评审，43% 的结构化走读，30% 的正规检视、5% 的其他。见图 1-10。

询问参与被调查人员“你们的组织对应用软件进行的动态测试种类？”。86% 的人回答进行系统测试，80% 的人员回答进行单元/模块测试，77% 的人员回答进行可接受性测试、

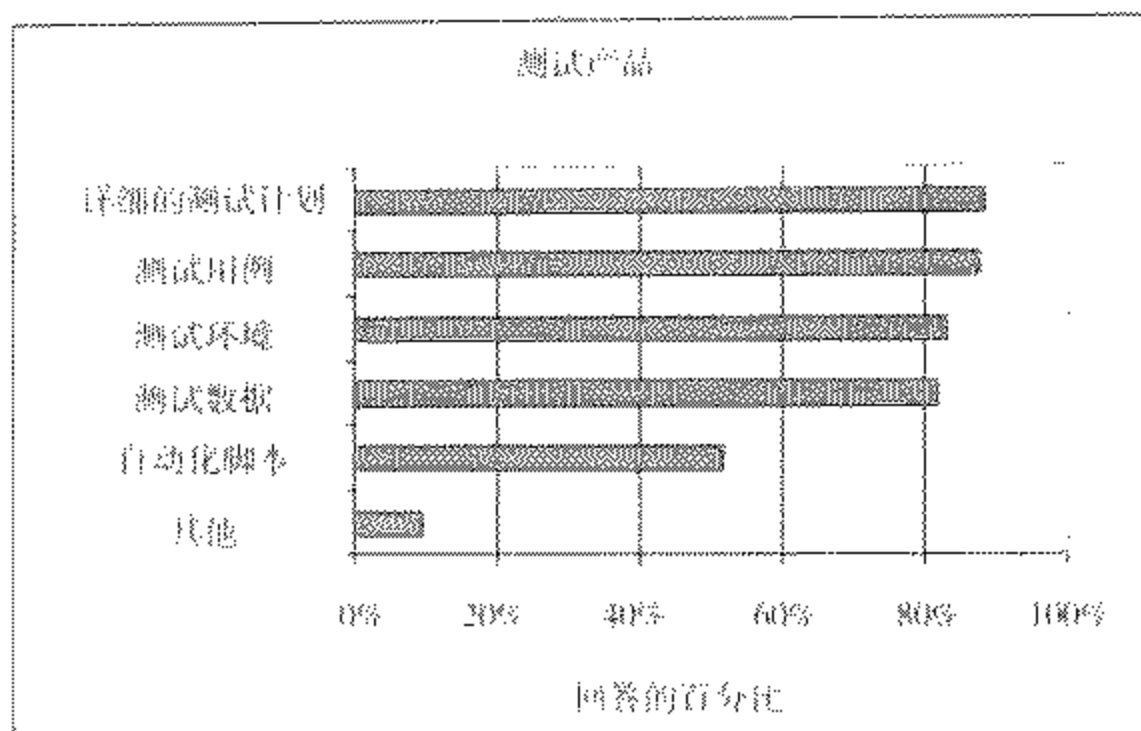


图 1-9 测试产品调查结果

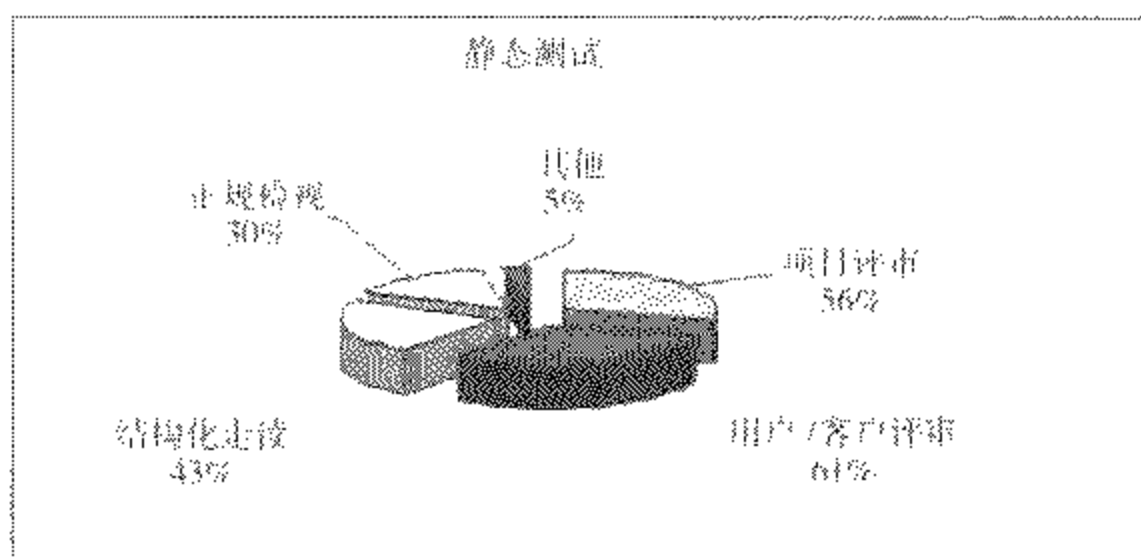


图 1-10 静态测试调查结果

75%的人员回答进行集成/联调测试，60%的人员回答进行压力/容量测试，46%的人员回答进行控制测试，9%的人员回答进行其他测试。1998年的结果调查表明在企业中有一种倾向，即进行更多种类型的测试，见图 1-11。

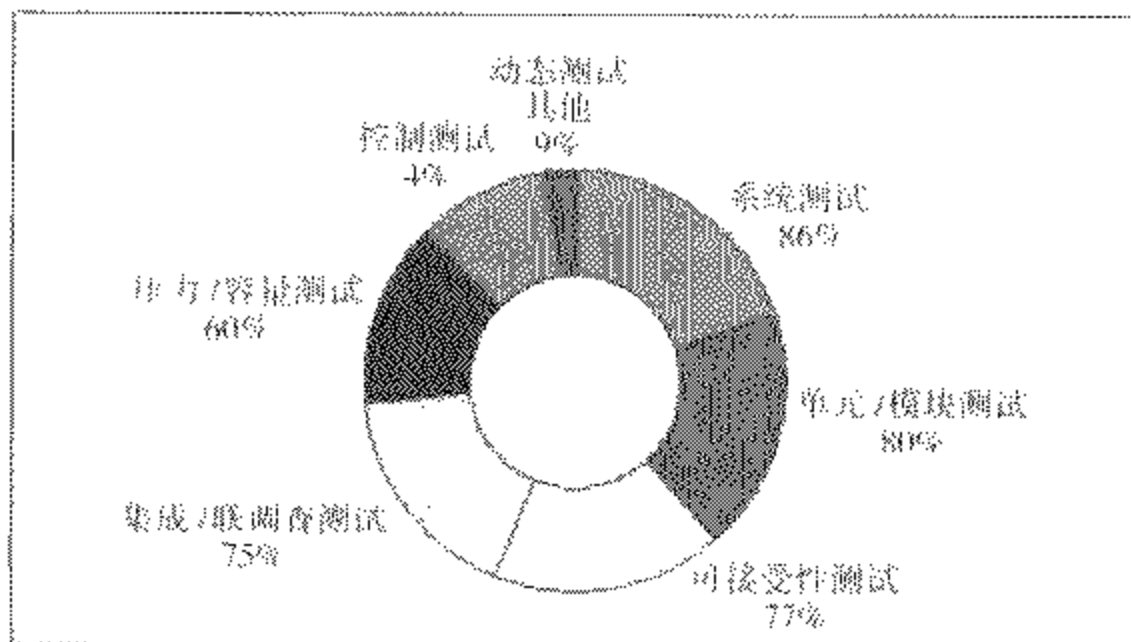


图 1-11 动态测试调查结果

询问参与被调查人员“测试完成决定软件可以准备交付”的问题时，肯定的回答比第二种原则的结果高出 20%。其他包括 49% 由用户决定，48% 由数据驱动决定，33% 由测试组决定，15% 使用了度量，13% 的其他，见图 1-12。

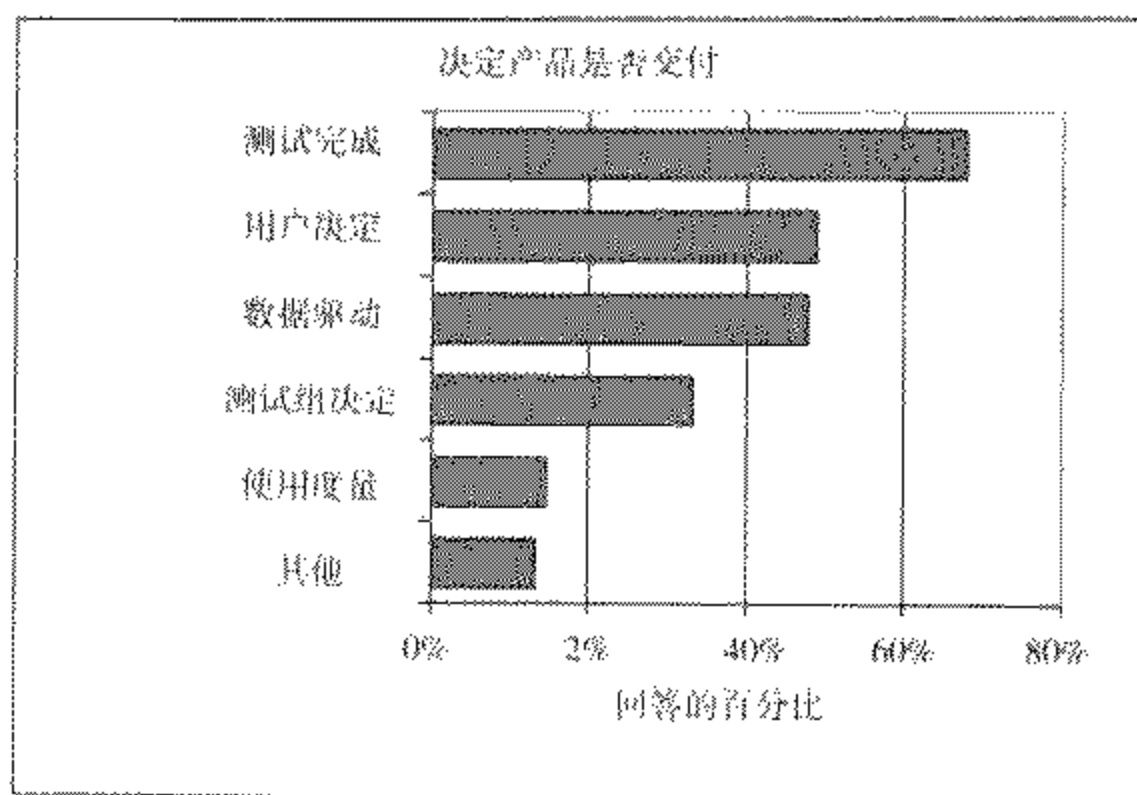


图 1-12 系统产品准备度调查结果

询问参与被调查人员“是否收集缺陷信息？”。91% 的人员回答是肯定的，见图 1-13。最常提到的度量指标是根据严重程度缺陷数量、打开的缺陷数、关闭的缺陷数、开发阶段的缺陷数、模块的缺陷数，每一千个功能点的缺陷数、不同缺陷产生原因的缺陷数、不同缺陷类型的缺陷数。

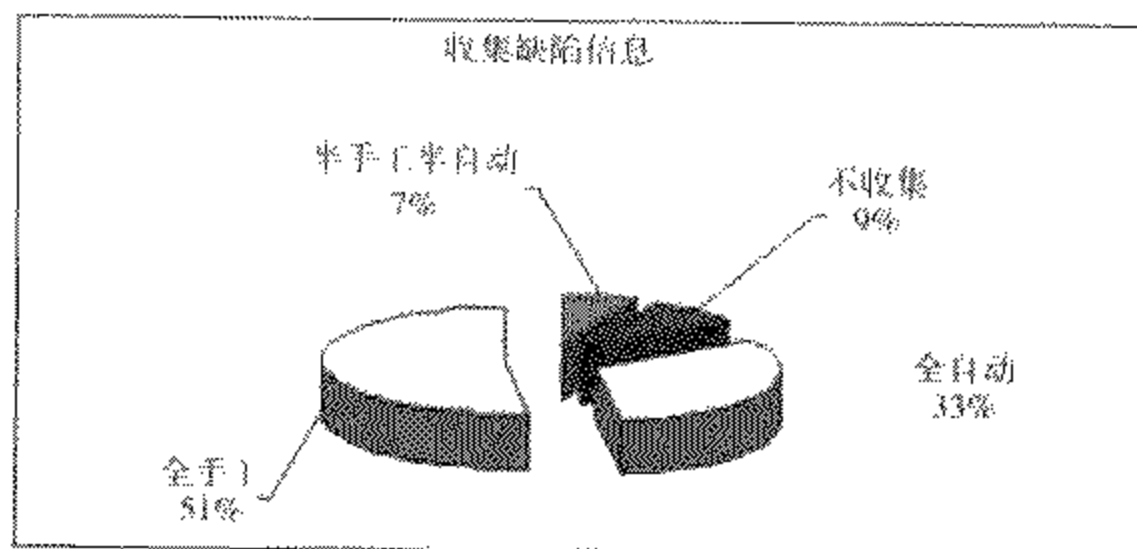


图 1-13 缺陷信息收集调查结果

询问参与被调查人员所在的组织是否准备测试状态报告？38% 的人员回答一直需要提交，53% 的人员回答有时提交，9% 的人员回答从不提交，见图 1-14。在报告中包含的信息有，74% 回答包含发现的缺陷，64% 包含缺陷的严重级别，58% 包含成功执行的测试用例，47% 包含被成功测试的缺陷，14% 包含没有被纠正的缺陷，13% 包含测试的成本，9% 是其他。

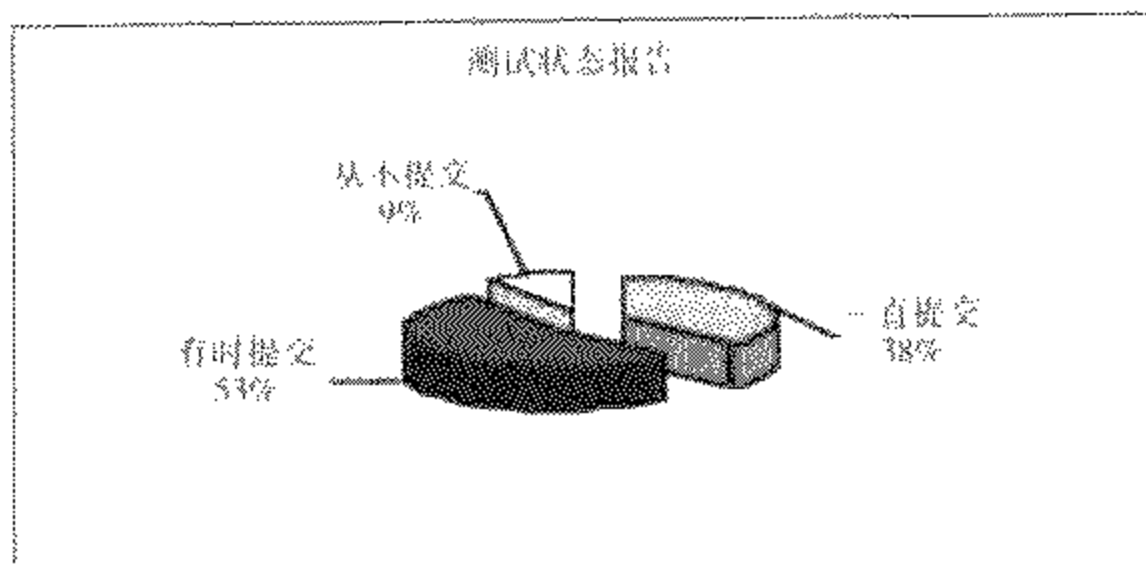


图 1-14 测试状态报告调查结果

1.6 软件测试中的误区

误区1 调试和测试是一样的

调试和测试是两个不同的过程，有着根本的区别。调试是一个随机的，不可重复的过程，它用于隔离和确认问题发生的原因，然后修改软件来纠正问题。而测试是一个有计划的，可重复的过程，它的目的是为了发现与预先定义的规格和标准不符合的问题。

误区2 测试组应当为质量保证负责^[4]

在一个组织中，测试组通常被安置为产品的最后防卫者角色，它是开发小组和用户之间的一道屏障。这使得测试组具有这样一个特点，即它拥有阻止产品被交付的权利。这个权利从其本身来说是令人沮丧的。测试组不能改进质量，更糟糕的是，权利的行使往往没有想象中那么容易。了解这一点，结合告诉开发人员质量是别人的事情这样一个不正确的动机，导致测试人员和测试组愤世嫉俗，并且把自己看成是一个受害者。我们已经从 Deming 和别人那里学到：只有当每个人在产品开发的每个阶段始终为他们的工作产品质量负责时，产品才能变得更好、更便宜^[4]。

误区3 过分依赖 Beta 测试。

Beta 测试似乎更能代表用户使用的测试用例——因为它们是用用户使用的。同时，用户报告的缺陷通常是用户最在乎的缺陷。但是，存在以下几个问题：

- 用户可能不具有代表性；
- 即使对于那些实际使用产品的用户来说，绝大部分不会严肃地去使用产品；
- Beta 用户不会及时地报告可用性方面的问题，他们会私下决定不买这个产品；
- Beta 用户经常不报告缺陷，尤其在不明确或者认为很明显，别人也会发现时；
- Beta 用户报告的缺陷经常是很简单，甚至是不可用的，你需要花费很长的时间处理一个用户缺陷。

误区4 把测试作为新员工的一个过渡工作

测试是一个系统性的工作，需要有很深的专业背景 and 知识。否则是无法做好测试工作的。因此把测试作为新手的一个过渡工作对测试本身是一种伤害，最终导致测试越来越薄弱，效果越来越差。

误区5 把不合格的开发人员安排做测试

有很多好的测试人员，但他们不是好的程序员。但是一个有某些习惯的差的程序员是不可能成为好的测试人员的。例如，那些经常会制造很多 Bug 的程序员是因为他们对细节不关注，这种人在测试中同样会漏过很多缺陷。

误区6 关注于测试的执行而忽略测试的设计

如果不关注测试设计，可能会遗漏很多特殊的用例，而这些用例往往也是开发人员没有考虑到的。因此一个好的测试必然是经过良好计划和良好设计的。不经过计划和设计的测试是不可控的、无序的。

误区7 测试自动化是万能的

测试自动化可以提高测试的效率，但不能提高测试的质量。由于自动化需要花费成本，因此只有那些经常需要执行的用例其自动化才能有效果。而且不是所有测试都可以或需要自动化的。当自动化到一定比例的时候，再提高自动化的成本将变得非常昂贵。业界有一个 2/8 原则，这同样适用于测试自动化，即花费 20% 的工作量可以完成 80% 的自动化工作，如果要完成其余的 20% 自动化，那么还需要再投入 80% 的成本^[7]。

误区8 测试是可以穷尽的

穷尽的测试是不可能的^[2]。作为测试人员经常会遇到项目经理要求测试人员必须去发现所有的错误。但是你怎么可能发现所有的错误呢？你能够完全地测试你的程序吗？当然不能，主要有下面几个原因^[27]：

- 你不可能测试程序的所有输入；
- 你不可能测试程序所有输入的组合；
- 你不可能测试通过程序的所有路径；
- 你不可能测试所有其他潜在的失败，例如有用户接口设计错误或者不完整需求分析而引起的错误。

误区9 测试是为了证明软件的正确性

测试无法证明软件是正确的，只能证明软件无法按照既定的规格和标准执行。测试的目的是尽可能地发现错误。

误区10 测试是枯燥乏味，缺乏创造力的工作

这在于你站在什么角度上看测试。一个好的测试需要经过计划，设计到执行。为了设计好的测试用例，你需要充分发挥你的想象力。对于一个缺乏想象力的测试人员来说，你可以胜任测试执行的工作，但是却无法设计出高质量的测试用例。无论你从事开发还是测试，都应当从工作中去寻找乐趣，不断地改进和完善自己。

1.7 本章小结

软件测试是伴随着软件和硬件的发展而逐步发展起来的，并且从最初的 Adhoc 测试发展到现在的全面测试理念。软件测试是一种检测手段，目的是为了寻找软件系统中的缺陷。在业界已经有越来越多的公司意识到了软件测试的重要性，并且在测试方面加大了投入。软件测试有很多误区，只有认识到了这些误区才能真正理解测试本身的含义，才能以正确的态度看待测试。

第2章 白盒测试和黑盒测试

白盒测试和黑盒测试是软件测试中的两大方法，传统的软件测试活动基本上都可以划到这两类方法当中。本章将对白盒测试概念、黑盒测试概念、白盒测试和黑盒测试的区别以及联系进行阐述，同时列举了常见的白盒测试方法和黑盒测试方法。通过本章的学习你应该掌握以下知识：

1. 什么是白盒测试？
2. 为什么要进行白盒测试？
3. 什么是黑盒测试？
4. 为什么要进行黑盒测试？
5. 白盒测试和黑盒测试有什么区别？
6. 常见的白盒测试技术有哪些？
7. 常见的黑盒测试技术有哪些？

2.1 白盒测试

2.1.1 什么是白盒测试

在测试类书籍中，白盒测试(White Box Testing)有多种叫法，如玻璃盒测试(Glass Box Testing)、透明盒测试(Clear Box Testing)、开放盒测试(Open Box Testing)、结构化测试(Structured Testing)、基于代码的测试(Code-Based Testing)、逻辑驱动测试(Logic-Driven Testing)等。白盒测试是一种测试用例设计方法。在这里，盒子指的是被测试的软件，白盒，顾名思义即盒子是可视的，你清楚盒子内部的东西以及里面是如何运作的。因此，白盒测试需要对系统内部的结构和工作原理有一个清楚的了解；并且基于这个知识来设计你的用例^{[9][10][11]}。图2-1是一个白盒测试的示意图。

使用白盒测试方法产生的测试用例能够：

- 保证一个模块中的所有独立路径至少被使用一次；
- 对所有逻辑值均需测试 true 和 false；
- 在上下边界及可操作范围内运行所有循环；
- 检查内部数据结构以确保其有效性。

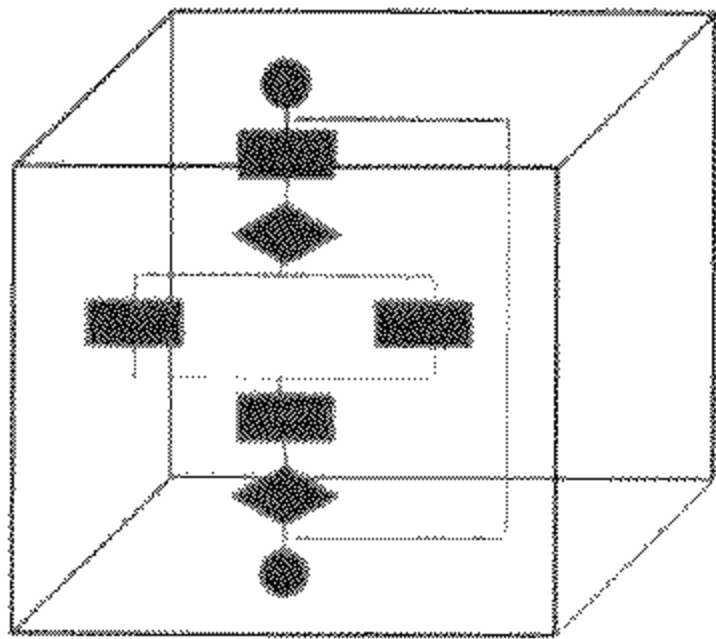


图 2-1 白盒测试示意图

2.1.2 为什么要进行白盒测试

“我们应该更侧重于保证程序需求的实现，为什么要花费时间和精力来担心（和测试）逻辑细节？”答案在于软件自身的缺陷：

- 逻辑错误和不正确假设与一条程序路径被运行的可能性成反比。当我们设计和实现主流之外的功能、条件或控制时，错误往往开始出现在我们工作中。日常处理往往能够被很好地理解，而“特殊情况”的处理则难于发现。
- 我们经常相信某逻辑路径不可能被执行，而事实上，它可能在正常的情况下被执行。程序的逻辑流有时是违反直觉的，这意味着我们关于控制流和数据流的一些无意识的假设可能导致设计错误，只有路径测试才能发现这些错误。
- 笔误是随机的。当一个程序被翻译为程序设计语言源代码时，有可能产生某些笔误，很多将被语法检查机制发现。但是，其他的在测试开始后才会被发现。笔误出现在主流路径上和不明显的逻辑路径上的几率是一样的。

正如 Beizer 所说：“错误潜伏在角落里，聚集在边界上”，而白盒测试更可能发现它^[16]。

2.1.3 白盒测试的常用技术

白盒测试技术一般可分为静态分析和动态分析两类技术^[14]。

1. 静态分析技术

静态分析是一种不通过执行程序而进行测试的技术^[15]。静态分析的关键功能是检查软件的表示和描述是否一致，没有冲突或者没有歧义。它瞄准的是纠正软件系统在描述、表示和规格上的错误。因此，是任何进一步测试执行的前提。静态分析覆盖程序语法的词汇分析，并研究和检查独立语句的结构和使用。主要有以下 3 种不同的程序测试可能性^[16]：

- 检查程序内部的完整性和一致性；
- 考虑预定义规则；
- 把程序和其相应的规格或文档进行比较。

虽然，有些软件工程师认为静态分析的特点是可以被自动执行，例如在一些特定工具的辅助下完成，像语法分析器、数据流分析器等。但是，用于测试的手工技术同样可以不需要程序的执行。图 2-2 列出了最重要的静态分析技术结构。这些技术在 1975 年—1994 年的软件工程协会的文献中可以找到。

语法分析器是一个基本的自动化静态分析工具，它把程序/文档文本分解成独立的语句。当在内部检查程序/文本时，语句的一致性会被检查。

当对两个文本在不同的语义级别上执行时，例如一个程序针对其规格文档，可以使用静态分析技术对程序的完整性和正确性进行评价^[16]。这个技术瞄准的是检测规格到

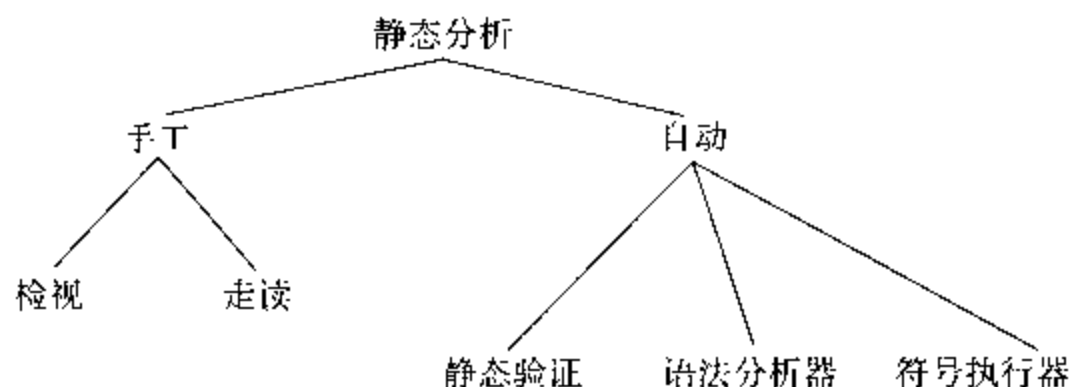


图 2-2 静态分析技术结构

程序实现之间转换的问题，称为静态验证。验证器需要有形式化的规格和规格的形式化定义。静态验证比较程序提供的实际值和规格文档中被预定义的目标值。然而，它不提供任何手段用于检查程序是否实际解决了给出的问题。验证过程的结果被描述成布尔语句，即一个语句要么是真，要么是假^[17]。静态验证的明显优点是它引导向目标和正确结果发展。但是，由于形式化规格是非常困难和耗时的，因此它一般用于那些要求高可靠性的软件。

另外一个静态分析技术是符号执行器。它在符号短语中分析一个程序在给定的路径上做了些什么事情。它模拟程序的执行，计算程序在不同位置上变量的值。符号执行器非常适合用于数学算法的分析。因为用于符号执行的程序开发是非常昂贵的，所以它一般用于测试数学程序，其成本/受益是可接受的。

静态分析技术中的一个最重要的手工技术是软件检视^{[18][19]}。最初提到检视技术的人是 Fagan^[20]，他认为在软件开发生命周期的多个阶段必须执行这个活动来改进软件质量。在检视中，代码或者工作产品的文档被使用预先定义好的检视规则进行检查^[18]。检视过程一般是根据检查表进行的。

走读是一个类似的同行评审过程，包括了程序的作者、测试人员、一个秘书和一个协调员。走读的参与者模拟计算机创建小部分数量的用例。它的目标是对源代码背后的逻辑和基本假设进行质疑，尤其是嵌入式程序中的接口。

关于同行评审的详细内容将在本书的第 16 章详细描述。

2. 动态分析技术

静态分析技术不需要软件的执行，而从动态分析本身来看更像是一个“测试”，因为它包含了系统的执行。当软件系统在模拟的或真实的环境中执行之前、之中和之后，对软件系统行为的分析是动态分析的主要特点。动态分析包含了程序在受控的环境下使用特定的期望结果进行正式的运行。它显示了一个系统在检查状态下是正确还是不正确。

在动态分析技术中，最重要的技术是路径和分支测试。在路径测试中，使程序能够执行尽可能多的逻辑路径。路径测试度量程序的最主要的质量特性是复杂度。分支测试需要程序中的每个分支至少被经过一次。分支测试中出现的问题可能会导致今后程序的缺陷。

当今，在软件开发过程中有许多动态分析工具。表 2-1 给出了对这些工具的分析^[18]。

表 2-1 动态分析工具

动态分析类型	工具的功能
测试覆盖率分析	测试白盒测试技术对代码的检测范围
跟踪	跟踪程序执行期间的所有路径, 例如所有变量的值等
调整	度量程序执行过程中使用的资源
模拟	模拟系统的部分, 例如, 无法获得的代码或硬件
断言检查	测试在复杂逻辑结构中是否某个条件已经被给出

3. 测试数据生成

在白盒测试中, 测试数据的选择和生成是一门重要的学科。最基本的方法是随机测试。对于随机测试, 大量的输入值被产生, 而不需要基于任何的结构和功能假设^[16]。当然还有两个成熟的测试数据产生方法, 它们是结构化测试和功能测试。结构化测试是根据程序的内部结构来指导测试数据的产生和选择。它在测试数据产生的时候会分析被测程序内部功能点, 因此避免了黑盒功能测试的局限性^[21]。Howden 描述的功能测试既考虑了系统的功能需求, 又考虑了一些重要的功能属性。这些属性是设计或实现的一部分, 没有被描述在需求文档中。在功能测试中, 一个程序被看成是一个函数, 并且考虑其输入值和输出值。现在市场上有一些测试用例生成的商用工具, 它们与特定的语言捆绑在一起。尤其对于嵌入式系统, 这些工具是非常有用的。因为它能模拟一个大系统的环境, 为每个可能的系统接口提供输入数据。

4. 覆盖率

在白盒测试中还有一个经常用到的技术是覆盖率技术。一方面, 覆盖率技术可以指导测试用例的设计; 另一方面, 可以通过覆盖率来衡量白盒测试的力度。白盒测试中经常用到的覆盖率是逻辑覆盖率, 主要有:

- 语句覆盖;
- 判定覆盖;
- 条件覆盖;
- 判定条件覆盖;
- 路径覆盖。

在实际测试中, 尤其在一些测试工具和理论文献当中, 还涉及其他一些类型的覆盖, 这些覆盖的详细解释将在本书的第 3 章介绍。

2.1.4 一个白盒测试的例子

在使用白盒测试时, 最理想的情况是希望能够执行到每个路径, 但这并不总是可能的。在表 2-2 中提供了一个示例。这个例子基本保证了每条路径至少被执行了一次^[22]。

表 2-3 白盒测试例子路径表

First if	Second if	if-else-if	Last if	Result
Income < 0	doesn't matter	doesn't matter	doesn't matter	negative income error
Income >= 0	NDependents <= 0	doesn't matter	doesn't matter	invalid dependents error
Income >= 0	NDependents > 0	Income < 10000	TaxTotal < 0	bracket 1 negative tax
Income >= 0	NDependents > 0	10000 <= Income < 50000	TaxTotal < 0	bracket 2 negative tax
Income >= 0	NDependents > 0	Income >= 50000	TaxTotal < 0	bracket 3 negative tax
Income >= 0	NDependents > 0	Income < 10000	TaxTotal >= 0	bracket 1
Income >= 0	NDependents > 0	10000 <= Income < 50000	TaxTotal >= 0	bracket 2
Income >= 0	NDependents > 0	Income >= 50000	TaxTotal >= 0	bracket 3

表 2-4 白盒测试例子测试套

Income	NDependents	Expected Result
-5	Doesn't matter	negative income error
0	0	invalid dependents error
100	1	0 (bracket 1, negative tax)
20000	11	0 (bracket 2, negative tax)
50000	100	0 (bracket 3, negative tax)
9000	1	130 (bracket 1)
15000	1	300 (bracket 2)
100000	1	3350 (bracket 3)

2.2 黑盒测试

2.2.1 什么是黑盒测试

黑盒测试(Black Box Testing)又叫功能测试(Functional Testing)。这是因为在黑盒测试中,主要关注于被测软件的功能实现,而不是内部逻辑。黑盒测试是与白盒测试截然不同的测试概念,也是在软件测试中使用得最早、最广泛的一类测试。在黑盒测试中,被测对象的内部结构、运作情况对测试人员是不可见的。测试人员对被测产品的验证主要是根据其规格,验证其与规格的一致性。就像对一台自动售货机,为了验证其能否自动售出货物,你可以指定需要购买的物品,塞入钱币,然后观测售货机能否输出正确的货物并找出正确的零钱。在这个过程中,你不需要关注自动售货机是如何判定钱币数额,如何选择货物,如何找出零钱等内部操作。这是白盒测试关注的范围,黑盒测试关注的是结果。图 2-3 是黑盒测试的一个示意图。

黑盒测试试图发现以下类型的错误:

- 功能错误或遗漏;
- 界面错误;

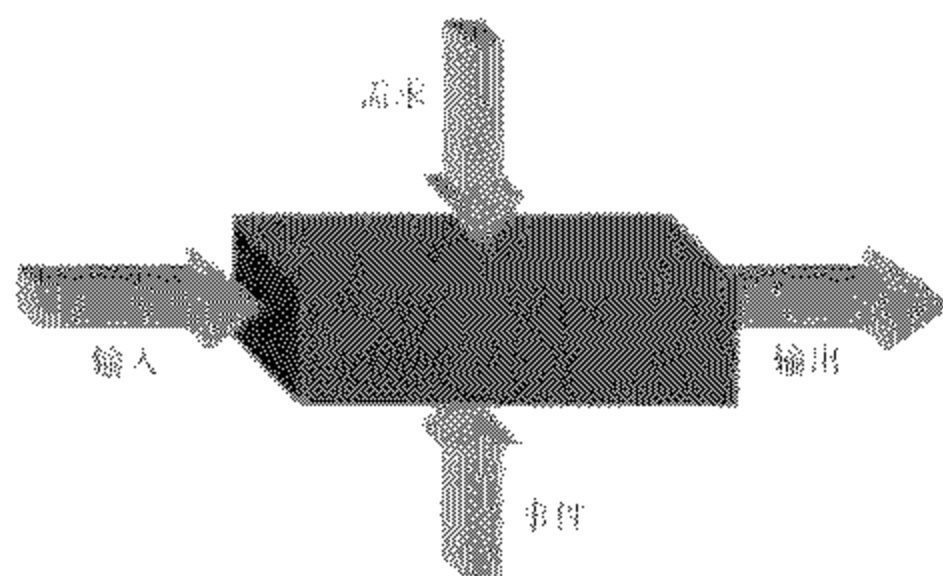


图 2-3 黑盒测试示意图

- 数据结构或外部数据库访问错误；
- 性能错误；
- 初始化和终止错误。

2.2.2 为什么要进行黑盒测试

既然我们已经做了白盒测试，为什么还要进行黑盒测试？这不是工作浪费吗？

白盒测试在测试的早期采用，而黑盒测试主要用于测试的后期。黑盒测试故意不考虑控制结构，而只注意信息域。黑盒测试并不是白盒测试的替代品，而是用于辅助白盒测试发现其他类型的错误。黑盒测试用于回答以下问题：

- 如何测试功能的有效性？
- 何种类型的输入会产生好的测试用例？
- 系统是否对特定的输入值尤其敏感？
- 如何分隔数据类的边界？
- 系统能够承受何种数据率和数据量？
- 特定类型的数据组合会对系统产生何种影响？

运用黑盒测试方法，可以导出满足以下标准的测试用例集：

- 所设计的测试用例能够减少，以达到合理测试所需的附加测试用例数；
- 所设计的测试用例能够告知某些类型错误的存在或不存在，而不是仅仅与特定测试相关的错误。

2.2.3 黑盒测试的常用技术

功能测试意味着测试数据的选择和测试结果的解释是以软件功能属性为基础的。黑盒测试不应当由程序作者来执行，因为他知道太多的程序内部知识^[10]。在新的测试方法中，软件系统在内部白盒测试完成后，由第三方来执行黑盒测试。

尽管黑盒测试是围绕著用户需求文档进行的，但是黑盒测试不一定必须要用户来参与测试。在绝大多数没有用户参与的黑盒测试中，最常见的测试有：功能性测试、容量测

试、安全性测试、负载测试、恢复性测试、标杆测试、稳定性测试、可靠性测试等。此外，有两个类型的测试必须要有用户参与，它们是外场测试和实验室测试^{[23][24]}。

1. 没有用户参与的黑盒测试

有两种不同途径的功能测试方法。一种是顺序测试每个程序特性或功能^[9]。另一种途径是一个模块一个模块的测试，即每个功能在其最先调用的地方被测试。

容量测试的目的是检测软件在处理海量数据时的局限性。容量测试能发现系统效率方面的问题，例如，不正确的缓冲区规模，消耗太多内存空间等。

负载测试检测系统在一个很短时间内处理一个巨大的数据量或执行许多功能调用上的能力。例如检测一个网站在某个时间段内接受 100 万用户的访问。

恢复性测试主要保证系统在崩溃后能够恢复外部数据的能力。系统能够完全恢复还是部分恢复这些数据？尤其对于需要高可靠性的系统。

标杆测试包含程序效率的测试。一段程序的有效性很大程度上依赖于硬件环境，因此标杆测试总是考虑软件和硬件的组合^[18]。然而，对于大部分软件工程师来说，标杆测试主要关注的是特定操作的量化数据^[25]。有时也考虑用户测试，比较不同软件系统作为标杆测试的有效性。

这些测试的详细内容将在本书后面章节进行详细描述。

2. 有用户介入的黑盒测试

对于用户介入的测试，在软件工程文献中提到的方法很少。比较实际的测试报告是大致在外场测试(类似 Beta 测试)和实验室测试(类似 Alpha 测试)之间的区别。

在外场测试中，观察用户在他们正常的工作地点使用软件的情况。除了一般的与可用性相关方面的特点外，外场测试对评价软件系统的可交互性特别有用。例如，系统工作的技术综合性如何等。此外，外场测试是阐明系统到达已有过程中的综合性能的仅有实际手段(即系统与实际环境的结合能力)。尤其在 NLP(自然语言处理)环境中，这个问题通常被低估。在实现一个翻译存储器综合性问题的一个典型的例子是，一个大的汽车制造商的语言服务。在此，主要的实现问题不是技术环境，而是实际上许多客户仍旧提交印刷件的定货单。这样原始文本和目标文本都无法被适当地组织和存储，最终导致单个的翻译器根本无法激起人们工作习惯的改变。

实验室测试一般用来评价系统的可用性方面的问题。由于实验室测试的高额成本，该测试一般只有在大型的软件机构才被进行，如 IBM、Microsoft 等。由于实验室测试给测试人员提供了许多技术可能性，因此其数据收集和分析比外场测试要容易得多。

2.2.4 一个黑盒测试的例子

在黑盒测试中，我们不依据代码来决定测试用例，而是在了解需要解决的问题的情况下，组织下面 4 个方面的测试数据：

- 易于计算的数据；
- 典型数据；

- 边界/极端数据;
- 假数据。

例如, 我们需要测试一个功能, 它使用二次方程式来决定 $ax^2 + bx + c$ 的两个根。为了尽可能简单, 我们假定只处理实数, 并且当两个根都是复数(负数的平方根)时, 打印错误信息。

想一下二次方程式的求根公式, 我们可以基于判别式 $b^2 - 4ac$ 来组织测试数据, 考虑上面的4个方面, 组织如表2-5~表2-8所示。

表 2-5 易于计算的数据(判别式是一个完全平方数)

a	b	c	根
1	2	1	-1, -1
1	3	2	-1, -2

表 2-6 典型数据(判别式是一个正数)

a	b	c	根
1	4	1	-3.73205, -0.267949
2	4	1	-1.70711, -0.292893

表 2-7 边界/极端数据(判别式是0)

a	b	c	根
2	-4	2	1, 1
2	-8	8	2, 2

表 2-8 假数据(判别式是负数或者 a 是0)

a	b	c	根
1	1	1	负数平方根
0	1	1	被0除

类似白盒测试, 必须使用上面的每组数据来测试你的代码, 如果每组数据都符合了, 那么你的代码就通过黑盒测试了。

2.3 白盒测试和黑盒测试的比较

从上面白盒测试和黑盒测试的例子来看, 我们可以发现白盒测试会考虑黑盒不会考虑的方面。同样, 黑盒测试也会考虑白盒测试不会考虑的方面。白盒测试只考虑测试软件产品, 它不保证完整的需求规格是否被满足。而黑盒测试只考虑测试需求规格, 它不保证实现的所有部分是否被测试到。黑盒测试会发现遗漏的缺陷, 指出规格的哪些部分没有被完成。而白盒测试会发现代理方面缺陷(faults of commission), 指出哪些实现部分是错误的。

白盒测试比黑盒测试成本要高得多。它需要在测试可以被计划前产生源代码, 并且在

确定合适的数据和决定软件是否正确方面需要花费更多的工作量。一个建议是，尽可能使用可获得的规格从黑盒测试方法开始测试计划。白盒测试计划应当在黑盒测试计划已经成功通过之后再开始，使用已经产生的流程图和路径的判定。路径应当根据黑盒测试计划进行检查并且决定和使用额外需要的测试。

一个白盒测试的失败会导致一次修改，这需要所有的黑盒测试被重复执行并且重新决定白盒测试路径。成本更低的手段是把测试过程考虑成是一个质量保证的过程而不是一个质量控制过程(质量保证内容请参考本书的第13章，质量控制就是传统意义上的测试)。早期工作产品的质量直接决定了后期测试的工作量，早期工作质量越好，那么后期测试能发现的错误越少，相应的测试和修改工作量就会减少，反之则增加。

2.3.1 白盒测试的优缺点

1. 优点

- 迫使测试人员去仔细思考软件的实现；
- 可以检测代码中的每条分支和路径；
- 揭示隐藏在代码中的错误；
- 对代码的测试比较彻底；
- 最优化。

2. 缺点

- 昂贵；
- 无法检测代码中遗漏的路径和数据敏感性错误；
- 不验证规格的正确性。

2.3.2 黑盒测试的优缺点

1. 优点

- 对于较大的代码单元来说(子系统甚至系统级)，黑盒测试比白盒测试效率要高；
- 测试人员不需要了解实现的细节，包括特定的编程语言；
- 测试人员和编码人员是彼此独立的；
- 从用户的视角进行测试，很容易被理解和接受；
- 有助于暴露任何规格不一致或有歧义的问题；
- 测试用例可以在规格完成之后马上进行。

2. 缺点

- 只有一小部分可能的输入被测试到，要测试每个可能的输入流几乎是不可能的；
- 没有清晰的和简明的规格，测试用例是很难设计的；
- 如果测试人员不被告知开发人员已经执行过的用例，在测试数据上会存在不必要

- 的重复;
- 会有很多程序路径没有被测试到;
- 不能直接针对特定程序段测试, 这些程序段可能很复杂(因此可能隐藏更多的问题);
- 大部分和研究相关的测试都是直接针对白盒测试的。

2.3.3 灰盒测试

白盒和黑盒这两类测试是从完全不同视角点出发的, 是完全对立的。这反映了事物的两个极端。两种方法各有侧重, 不能替代。但是, 在现代测试理念中, 这两种测试方法往往不是截然分开的。一般地, 在白盒测试中交叉使用黑盒测试的方法; 在黑盒测试中交叉使用白盒测试的方法。灰盒测试就是这类介于白盒测试和黑盒测试之间的测试。最常见的灰盒测试是集成测试。关于集成测试的详细内容将在本书的第7章讲解。

2.4 本章小结

黑盒测试和白盒测试都是测试设计的方法。黑盒测试把系统理解为一个“内部不可见的盒子”, 因此不需要明白它的内部结构。黑盒测试一般关注的是对功能需求的测试。白盒测试设计允许你观察“盒子”内部, 让你了解其内部结构和运作原理, 并使用对这些知识的了解来指导测试用例的设计。为了完全测试一个软件, 不可或缺任一种测试。现今的测试技术也越来越趋向于多样化。一个产品(尤其是需要高可靠性的产品)在其概念分析阶段直到最后交付给用户期间往往要经过各种静态的、动态的、白盒的和黑盒的测试。任何一个级别的测试(单元、集成、系统等)都可以使用任何一种测试设计方法。

第3章 测试覆盖率

覆盖率是软件测试经常会用到的一个概念，本章主要就这个内容进行阐述。通过本章的学习，你应当掌握以下几个问题：

1. 什么是覆盖率？
2. 常见的逻辑覆盖有哪些类型？它们各表示什么含义？
3. 覆盖率对软件测试有什么作用？
4. 如何正确地使用覆盖率？

最近的许多研究表明，系统可靠性能从可测试性的角度来衡量。缺陷排除率和代码覆盖率之间的关系在一定程度上呈现为线性函数。测试用例设计不能一味追求覆盖率，因为测试成本随覆盖率的增加而增加。

3.1 覆盖率概念

覆盖率是用来度量测试完整性的一个手段。覆盖率的种类有很多，我们经常接触到的覆盖率大体上可以划分为两大类：逻辑覆盖和功能覆盖。现在有越来越多的测试工具能够支持覆盖率测试。但是，这些度量本身并不包含测试技术，它们只是测试技术有效性的一个度量。

覆盖率可以通过一个比率公式来表示：

$$\text{覆盖率} = (\text{至少被执行一次的 item 数}) / \text{item 的总数}$$

公式中，假设要对 item 的覆盖情况进行计算^[28]。覆盖率对于软件测试有着非常重要的作用。通过覆盖率数据，可以知道测试得是否充分，测试的弱点在哪些方面，进而指导我们设计能够增加覆盖率的测试用例。这样就能够有效地提高测试质量，避免设计无效用例。

3.2 常见的逻辑覆盖

覆盖率中最常见的是逻辑覆盖率(Logical Coverage)，也叫代码覆盖率(Code Coverage)或结构化覆盖率(Structural Coverage)。逻辑覆盖属于白盒测试的范畴。我们经常接触到的逻辑覆盖包括：语句覆盖、判定覆盖、条件覆盖、判定条件覆盖、路径覆盖等。

3.2.1 语句覆盖

语句覆盖(Statement Coverage)的含意是，在测试时，首先设计若干个测试用例，然后

运行被测程序，使程序中的每个可执行语句至少执行一次。所谓“若干个”，自然是越少越好。语句覆盖率的公式可以表示如下：

$$\text{语句覆盖率} = (\text{至少被执行一次的语句数量}) / (\text{可执行的语句总数})$$

语句覆盖是逻辑覆盖中最简单的覆盖。从度量的角度看，必须始终跟踪执行语句的情况，然后同所有可执行的语句进行比较。因此，语句覆盖是比较适合自动化的，也是比较容易理解的。这使对不完整的语句覆盖进行分析变得容易。几乎对所有代码达到100%的语句覆盖率是现实的。然而，语句覆盖不是测试完整性方面的一个好的度量。我们来看下面一段代码：

例1A：

```
1. if CONDITION then
2.   DO_SOMETHING;
3. end if;
4. ANOTHER_STATEMENT;
```

对于例1A，要达到完全的语句覆盖，只需要一个用例就可以了，这个用例中CONDITION为真。但是，这个用例无法区别例1A的代码和例1B的代码。

例1B：

```
1. null;
2. DO_SOMETHING;
3. null;
4. ANOTHER_STATEMENT;
```

由于语句覆盖的简单性，因此用于语句覆盖的测试数据是非常容易维护的。

3.2.2 判定覆盖

判定覆盖(Decision Coverage)也叫分支覆盖(Branch Coverage)。它的含义是，在测试的时候设计若干测试用例，运行被测程序，使得程序中的每个判断至少取真分支和假分支一次，即判断的真假值均曾被满足。判定覆盖率的公式如下：

$$\text{判定覆盖率} = (\text{判定结果被评价的次数}) / (\text{判定结果的总数})$$

参见下面的例2A。要满足其判定覆盖必须要有两个用例：

例2A：

```
1. if CONDITION then
2.   DO_SOMETHING;
3. else
4.   DO_SOMETHING_ELSE;
5. end if;
Test  CONDITION
1     True
2     False
```

从这个例子可以发现，判定覆盖同样满足语句覆盖。并不是所有的判定条件都如此简单。判定条件还存在于case语句(switch语句)和循环语句中。但是，这些并不是自动化的

障碍。

判定覆盖也是直接针对代码的,这使得判定覆盖容易理解。对实际的代码调查表明,没有不可实现的判定结果。因此,对所有被分析的模块来说,要达到 100% 的判定覆盖是可能的。类似于语句覆盖,设计用于判定覆盖的测试数据也是易于维护的。对于类似例 2A 的代码的例 2B,不需要对例 2A 的测试数据做任何修改就可以用于例 2B,达到判定覆盖。

例 2B:

```
1. if not CONDITION then
2.   DO_SOMETHING_ELSE;
3. else
4.   DO_SOMETHING;
5. end if;
```

判定覆盖的缺点是很明显的。当复杂的条件用于控制分支时,判定覆盖就显得不足了。如例 2C,使用两个测试用例就可以达到判定覆盖,但却无法进行完整的测试。

例 2C:

```
1. if A and B then
2.   DO_SOMETHING;
3. else
4.   DO_SOMETHING_ELSE;
5. end if;
```

Test	A	B
1	True	True
2	False	True
	Untested	
	Ture	False
	False	False

对于复合条件,两个或多个条件项的组合可能会导致一个特定的分支被执行,判定覆盖会在其一个组合中被测试到。因此判定覆盖的完整性虽然比语句覆盖高,但是却不如条件覆盖。同时,复合条件是代码错误的一个根源。

3.2.3 条件覆盖

条件覆盖(Condition Coverage)的含义是,设计若干测试用例,执行被测程序后,要使每个判断中每个条件的可能取值至少满足一次。条件覆盖率的公式如下:

条件覆盖率 = (条件操作数值至少被评价一次的数量)/(条件操作数值的总数)

这里涉及到操作数的概念。从赫尔斯梯德软件度量来看^[30],软件中的元素包括操作数(Operand)和操作符(Operator)。例如下面列出的都是操作数:

常量:

45	(decimal integer, 整数)
0x53 或 0X53	(hexadecimal, 十六进制数)
0177	(octal, 八进制数)

2324.57	(real, 实数)
'c', 或 '\432'	(character, 字符)
"hello"	(string, 字符串)

变量:

id1	(simple type, 简单类型)
id1[*]	(array type, 数组类型)
id1.id2.	(record type, 记录类型)
id1 -> id2	(field pointer type, 指针类型)
id1[][] id1.id2[] id1[] -> id2	(combinasion, 联合)

下面列出的都是操作符:

```
IF - ELSE , WHILE(), DO WHILE()
RETURN , FOR(;;), SWITCH , BREAK , CONTINUE
GOTO label
,
;      (除了在 FOR 的表达式中)
( )   (used to delimit an expression)
[ ]   (array subscript)
```

存在于表达式中的操作符有:

- 单一操作符:

-	(负号)
++, --	(递增或递减)
!	(取反)
sizeof	(取规模)
*	(取值)
&	(取地址)
~	(complement of 1)

- binary operators:

+, -, *, /, %	(算术操作符)
<<, >>, &, , ^	(位操作符)
>, <, <=, >=, ==, !=	(比较操作符)
&&,	(逻辑操作符)

- 条件操作符: ?:

- 赋值操作符:

= += -= *= /= % > >= < <= &= *= |=

条件操作数很容易从设计或代码中被确认。这有助于自动化,并使条件操作数覆盖易于理解和维护。由于考虑每个判定条件的真/假,因此条件覆盖要比分支覆盖强,我们来看下面这个例子:

例 3A:

```
IF ( A > 1 ) AND ( B = 0 )   THEN   X = X / A
IF ( A = 2 ) OR ( X > 1 )   THEN   X = X + 1
```

在上述程序段中，第一个判断应考虑到：

$A > 1$ ，取真值，记为 T_1 。

$A > 1$ ，取假值，即 $A \leq 1$ ，记为 F_1 。

$B = 0$ ，取真值，记为 T_2 。

$B = 0$ ，取假值，即 $B \neq 0$ ，记为 F_2 。

第2个判断应考虑到：

$A = 2$ ，取真值，记为 T_3 。

$A = 2$ ，取假值，即 $A \neq 2$ ，记为 F_3 。

$X > 1$ ，取真值，记为 T_4 。

$X > 1$ ，取假值，即 $X \leq 1$ ，记为 F_4 。

因此，对该例使用3个测试用例就可以达到条件覆盖的目标，见表3-1。

表3-1 满足条件覆盖和判定覆盖的用例

测试用例	A	B	X	覆盖条件
CASE 1	2	0	3	T_1, T_2, T_3, T_4
CASE 2	1	0	1	F_1, T_2, F_3, F_4
CASE 3	2	1	1	T_1, F_2, T_3, F_4

如果应用判定覆盖的概念，从上面这组用例发现，其判定覆盖也被满足。但是，满足条件覆盖是否就一定满足判定覆盖呢？回答是“未必”。请看表3-2的一组用例。

表3-2 只满足条件覆盖的用例

测试用例	A	B	X	覆盖条件
CASE 4	1	0	3	F_1, T_2, F_3, T_4
CASE 5	2	1	1	T_1, F_2, T_3, F_4

显然，该组用例满足了条件覆盖的要求，但却不能满足判定覆盖要求。因此，应当还有比条件覆盖更强的覆盖。

3.2.4 判定条件覆盖

判定条件覆盖 (Decision Condition Coverage) 也叫分支条件覆盖 (Branch Condition Coverage, B-C Coverage)。其含义是，设计足够多的测试用例，使得判断中每个条件的所有可能值 (为真为假) 至少出现一次，且每个判断本身的判定结果 (为真为假) 也至少出现一次。判定条件覆盖率的公式如下：

$$\text{判定条件覆盖率} = \frac{\text{条件操作数值或判定结果至少被评价一次的数量}}{\text{条件操作数值总数} + \text{判定结果总数}}$$

类似于条件覆盖和分支覆盖，判定条件覆盖也便于自动化，且易于理解和维护。同时，判定条件覆盖是一个比判定覆盖和条件覆盖更强的覆盖。对于例3A，分析两个判定中4个条件的组合情况如下：

- ① $A > 1, B = 0$ 记为 T_1, T_2
- ② $A > 1, B \neq 0$ 记为 T_1, F_2
- ③ $A \leq 1, B = 0$ 记为 F_1, T_2
- ④ $A \leq 1, B \neq 0$ 记为 F_1, F_2
- ⑤ $A = 2, X > 1$ 记为 T_3, T_4
- ⑥ $A = 2, X \leq 1$ 记为 T_3, F_4
- ⑦ $A \neq 2, X > 1$ 记为 F_3, T_4
- ⑧ $A \neq 2, X \leq 1$ 记为 F_3, F_4

由此, 表 3-3 中的一组用例就可以满足判定条件覆盖。

表 3-3 满足判定条件覆盖用例

测试用例	A	B	X	覆盖组合号		覆盖条件
CASE 1	2	0	3	①	⑤	T_1, T_2, T_3, T_4
CASE 2	2	1	1	②	⑥	T_1, F_2, T_3, F_4
CASE 3	1	0	3	③	⑦	F_1, T_2, F_3, T_4
CASE 4	1	1	1	④	⑧	F_1, F_2, F_3, F_4

仔细分析例 3A 的执行路径(见图 3-1)就会发现表 3-3 的用例只能覆盖路径 ace, abe, abd, 而未能覆盖路径 acd。可见判定条件覆盖还不是完整的覆盖, 无法满足对程序的完整测试。因此, 还需考虑路径覆盖。

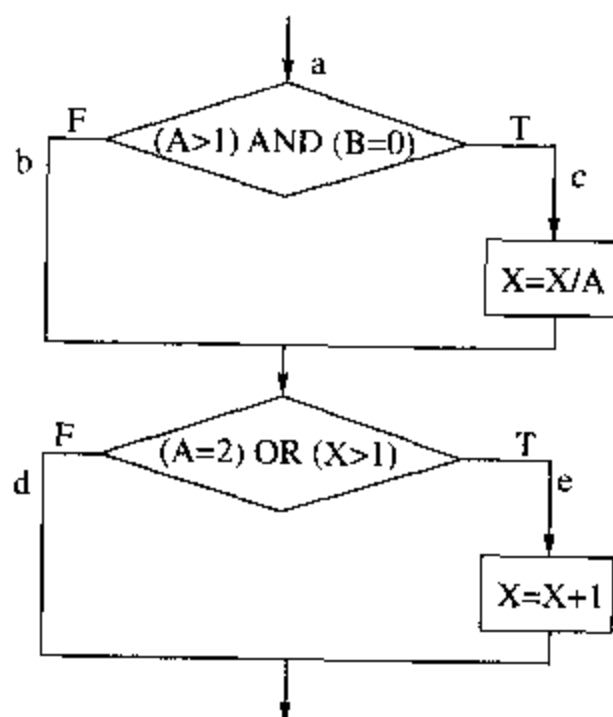


图 3-1 例 3A 流程图

3.2.5 路径覆盖

路径覆盖(Path Coverage)的含义是, 设计足够多的测试用例, 要求覆盖程序中所有可能的路径。路径覆盖率的公式如下:

$$\text{路径覆盖率} = (\text{至少被执行到一次的路径数}) / (\text{总的路径数})$$

对于例 3A, 如果要达到路径覆盖, 可以使用表 3-4 列出的一组用例。

表 3-4 满足路径覆盖的用例

测试用例	A	B	X	覆盖路径
CASE 1	2	0	3	a c e
CASE 2	1	0	1	a b d
CASE 3	2	1	1	a b c
CASE 4	3	0	1	a c d

路径覆盖企图在程序中寻找完全的路径数。例如，若一个模块包含一个循环，那么对于循环的第一次迭代、第二次迭代、直到第 n 次迭代都是独立的路径。因此，要获得 100% 路径覆盖的完整性是很高的。在实际代码中，一个较为复杂的程序包含的路径数也是相当庞大的。因此，要完全达到路径覆盖是不可行的，实现自动化路径覆盖也是困难的。

尽管路径覆盖比判定条件覆盖更强，但是路径覆盖并不能包含判定条件覆盖。如在例 4A 中，只要两个用例就可以达到路径覆盖：(True, True), (True, False)。但显然没有达到条件覆盖。

例 4A:

```

1. if A and B then
2.   DO_SOMETHING;
3. else
4.   DO_SOMETHING_ELSE;
5. end if;
```

3.2.6 逻辑覆盖小结

在实际应用中，还有许多种结构化覆盖的例子，具体参见本章 3.5 节内容。虽然，结构化覆盖率可以作为测试完整性的一个度量；但是，即使达到了 100% 的结构化覆盖率，还是无法保证程序的正确性。这一严酷的事实对热心测试的程序员似乎是一个严重的打击。然而要记住，测试的目的并非要证明程序的正确性，而是要尽可能找出程序中的错误。事实上，现在还不存在一种十全十美的测试方法，能够发现所有的错误。想要撒下儿网就把湖中的鱼全都捕上来是做不到的；也就是说，软件测试是有局限性的。

3.3 功能覆盖率

功能覆盖(Function Coverage)属于黑盒测试范畴。在实际测试中，涉及到的覆盖率一般都是结构化覆盖率，与黑盒相关的覆盖率比较少。

功能覆盖中最常见的是需求覆盖，其含义是通过设计一定的测试用例，要求每个需求点都被测试到。需求覆盖率的公式是：

$$\text{需求覆盖率} = (\text{被验证到的需求数量}) / (\text{总的需求数量})$$

在黑盒测试中，还有一种覆盖称为接口覆盖，又叫入口点覆盖。要求通过设计一定的

用例使系统的每个接口都被测试到。

由于黑盒测试把被测系统理解为一个黑盒，测试时输入测试数据，然后判定输出结果是否与期望结果一致。根据测试可以获得输入数据的覆盖情况，即通过设计一定的用例，要求每种数据情况都被测试到。功能测试覆盖方面的自动化工具比较少。

3.4 面向对象的覆盖率

结构化覆盖率用来度量测试的完整性已经被大家所接受。但是，这一技术在面向对象领域却遇到了挑战。因为传统的结构化度量没有考虑到面向对象的一些特性，如多态性、继承性和封装性等。先看图 3-2 中的例子。

```
class Base {
public:
    void foo()      { ... helper (); ... }
    void bar()      { ... helper (); ... }
private:
    virtual void helper() { ... }
};
class Derived : public Base {
private:
    virtual void helper() { ... }
};
void test_driver() {
    Base base;
    Derived derived;
    base.foo();           //Test Case A1
    derived.bar();        //Test Case A2
}
```

图 3-2 一个继承例子

在这个例子中，测试用例 A1 调用了 Base 对象的 Base::foo() 方法，该方法又调用了 Base 对象的虚函数 helper()，即 Base::helper()。

在测试用例 A2 中，由于方法 bar 没有在 Derived 中被重载，因此仍旧继承方法 Base::bar()，该方法又调用了派生类 Derived 的 helper() 方法，即 Derived::helper()。

这是否已经完整测试了程序呢？

从传统的结构化覆盖率来看，它已经 100% 地执行了所有语句和分支，甚至路径。但是，我们很清楚地知道，该程序并没有被完全测试。因为，还没有测试 Base::bar() 和 Base::helper() 或者 Base::foo() 和 Derived::helper() 之间的接口。显然，Base::bar() 能够与 Base::helper() 很好地工作并不意味着 Base::bar() 能够很好地和 Derived::helper() 工作。图 3-3 显示了继承方法并没有完全被最初的测试用例完全覆盖。

为了达到 100% 的覆盖，我们必须加强测试，图 3-4 给出的用例可以满足要求。

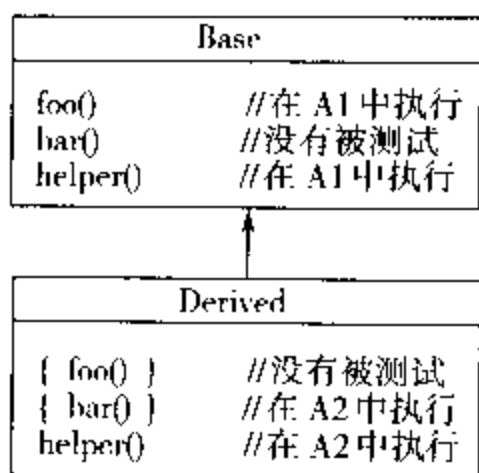


图 3-3 没有被完全测试到的继承方法

```

void better_test_driver() {
    Base base;
    Derived derived;
    base.foo();           //Test Case B1
    base.bar();           //Test Case B2
    derived.foo();        //Test Case B3
    derived.bar();        //Test Case B4
}

```

图 3-4 完全测试继承的测试用例

其中, 继承覆盖情况如图 3-5 所示。

由上面的例子可以发现, 传统的结构化覆盖必须被加强, 以满足面向对象特性, Cantata ++ (Cantata 是 IPL 的一个用于 C/C++ 语言单元测试的著名工具, 类似的工具还有 AdaTest, RTRT 等) 扩展传统的结构化覆盖率达到上下文覆盖 (Context Coverage)^[29]。上下文覆盖是一种收集被测试软件如何执行数据的方法。

上下文覆盖可以应用到面向对象领域处理诸如多态、继承和封装的特性。同时, 也可以被扩展用于多线程应用。通过使用这些面向对象的上下文覆盖, 结合传统的结构化覆盖的方法就可以保证代码的结构被完整地执行, 增强我们对被测软件质量的信心。

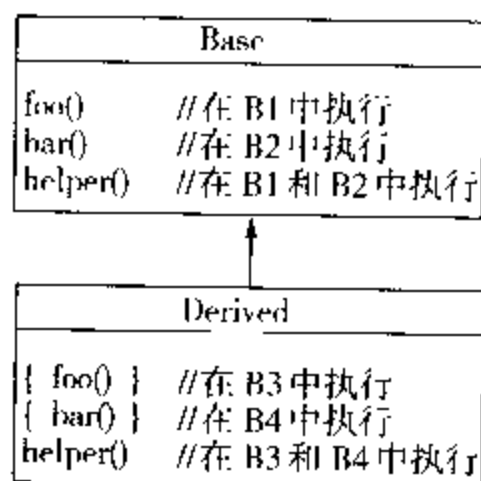


图 3-5 继承覆盖情况

有 3 个面向对象上下文覆盖的定义, 它们分别是: 继承上下文覆盖 (Inheritance Context Coverage), 该覆盖用于度量在系统中的多态调用被测试得有多好; 基于状态的上下文覆盖 (State-Based Context Coverage), 该覆盖用于改进对带有状态依赖行为的类的测试; 已定义用户上下文覆盖 (User-Defined Context Coverage), 允许将上下文覆盖的方法应用于传统结构化覆盖率无法使用的地方, 例如多线程应用。

3.4.1 继承上下文覆盖

继承上下文覆盖不是单个的度量。它是一种扩展传统结构化覆盖来考虑方法被继承时的额外接口。继承上下文覆盖提供了一个可替代的度量定义。它考虑在每个类的上下文内获得的覆盖率级别。在继承上下文覆盖定义中, 基类的方法在其上下文空间中的执行是完全独立于基继承类的上下文空间; 同样, 继承类的方法在其上下文空间中的执行也独立于其基类的上下文空间。为了获得 100% 继承上下文覆盖, 代码必须在每个适当的上下文内被完全执行。

考虑一个特殊的例子。在一个上下文内基于判定覆盖的继承上下文覆盖率, 可简单地用上下文内执行到的判定分支数据量除以程序内判定的总数来表示。

对于例行程序总的继承上下文判定覆盖被定义成对应该例行程序的每个上下文内继承上下文判定覆盖的平均数。对于一个被定义在基类中的例行程序, 其合适的上下文对应于

基类连同那些未加修改继承该例行程序的派生类。基类的例行程序不必要在对该例行程序重载的派生类中测试。

对于一个例行程序，其总的继承上下文判定覆盖率公式如下：

继承上下文判定覆盖率 =

(累加每个上下文内执行到的判定分支数)/(上下文数 × 上下文内的判定分支总数)

结合标准的结构化覆盖率，其对应的继承上下文覆盖也可以类似地被定义。

3.4.2 基于状态的上下文覆盖

在绝大多数面向对象系统中，存在许多类。最好把它们描述为状态机。这些类的对象可以存在于众多不同状态中的任何一种，而且每个类的行为在每个可能的状态中其性质是不同的——类的行为依赖于状态。如何测试这种类成了传统覆盖率的一个难题。

1. 基于状态的类

考虑有状态依赖行为的类，如一个有边界的栈。一个有边界的栈可能有3种状态：“空状态”、“部分满状态”，或者“满状态”。栈的行为在不同的状态上是不同的。例如，对于Pop()方法，它删除栈顶部元素并把该元素返回给调用者。该方法在“部分满状态”或者“满状态”时是正常的；而在“空状态”时，该方法抛出一个异常，并且使栈不做任何改动，仍旧保留在“空状态”上。

图3-6是有边界栈类的类接口定义。

```
class BoundedStack {
public:
    BoundedStack(size_t maxsize);
    ~BoundedStack();
    void push(int);
    int pop();
    struct underflow: std::exception {};
    struct overflow: std::exception {};
};
```

图3-6 有边界栈的类接口

2. 入口点覆盖

在图3-6的例子中，如果从黑盒测试角度出发，可以使用图3-7的一组用例来达到接口的覆盖(Interface Coverage)，或者入口点覆盖(Entry-Point Coverage)。

```
int test_driver() {
    BoundedStack stack(2);
    stack.push(1);
    stack.pop();
    // 这里隐含了析构函数的调用
};
```

图3-7 符合接口覆盖的用例

很显然,上面的用例无法完全测试该栈,因为栈不会满,也不会有异常抛出。

3. 白盒覆盖

对于使用白盒覆盖,例如判定覆盖,同样希望要达到 100% 的判定覆盖率。不幸的是,获得这个覆盖有不利的因素。其中一个因素是在代码中存在这样一些判定,它们不对应于公共接口的特性。典型的例子是错误处理代码和保护性程序代码。对其实现 100% 的判定覆盖难以达到。

另一个更重要的因素是,传统的结构化覆盖率对于确定哪些代码被丢失方面无能为力。例如在 BoundedStack 类中,实现人员忘了在 Pop 方法中考虑空状态条件的检查。这时,即使达到 100% 判定覆盖也无法发现这个错误。

4. 基于状态转移图

事实上,我们可以在不了解类内部细节的基础上写出比传统结构化覆盖率度量更好的测试集来。请参见图 3-8 的 UML 状态迁移图。该图显示了类行为的变化,它依赖于类当前的状态。

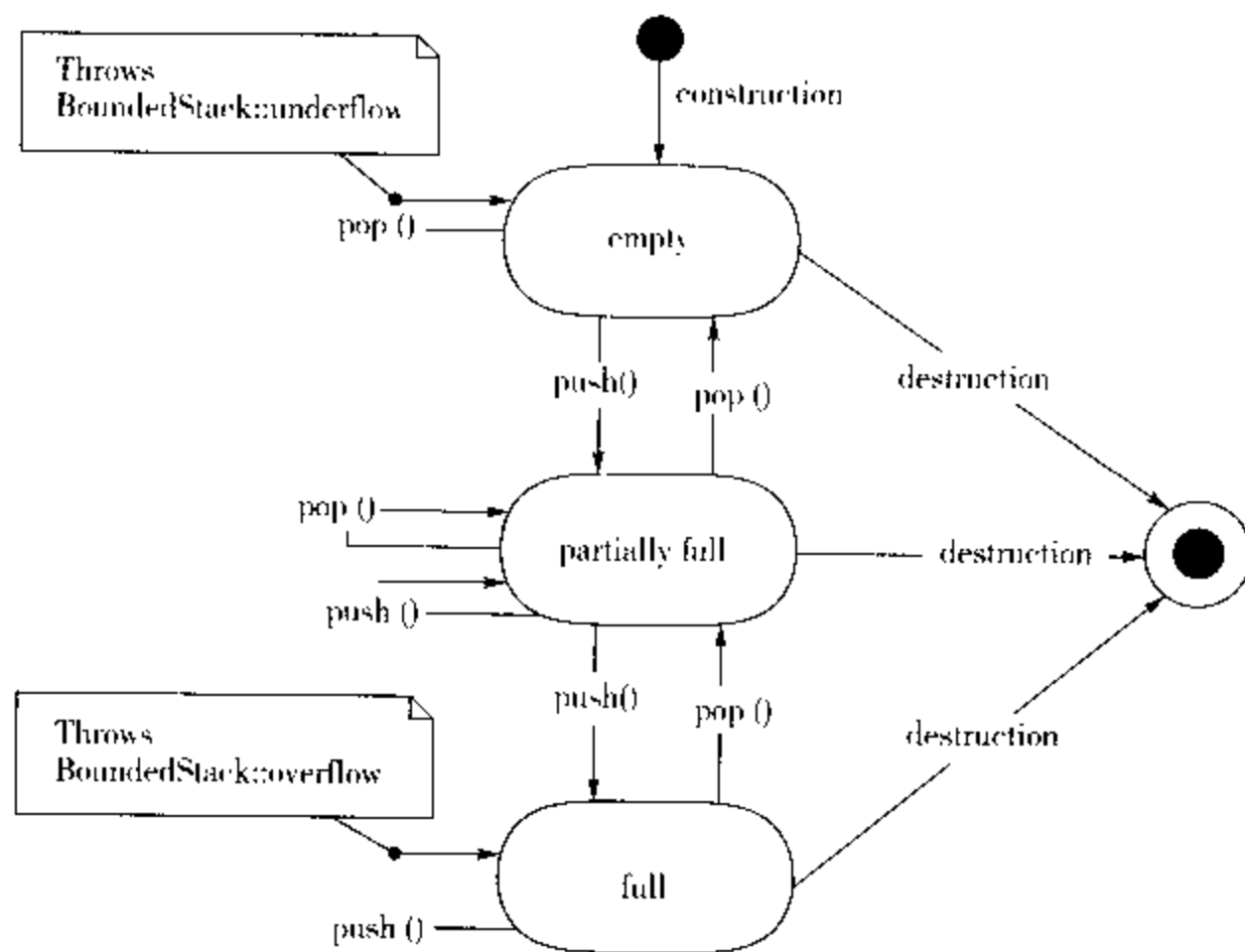


图 3-8 BoundedStack 的状态迁移图

根据该状态迁移图,可以设计一个能完全执行 BoundedStack 类的测试集。测试设计目标是执行每个可能状态的每种方法。在图 3-9 的测试集中, push() 和 pop() 方法在“空状态”、“部分满状态”和“满状态”中都被调用。

那么上面这个例子能否完整地测试 BoundedStack 类呢?可以用基于状态的上下文覆盖进行度量。

```
int better_test_driver() {  
    BoundedStack stack(2);    // 当空的时候执行 push()  
    stack.push(3);            // 当部分满的时候执行 push()  
    stack.push(1);  
    try { stack.push(9); }     // 当满的时候执行 push()  
    catch (BoundedStack::overflow) { } // 期望抛出异常  
    stack.pop();              // 当满的时候执行 pop()  
    stack.pop();              // 当部分满的时候执行 pop()  
    try { stack.pop(); }       // 当空的时候执行 pop()  
    catch (BoundedStack::underflow) { } // 期望抛出异常  
    // 这里隐藏了析构函数的调用  
};
```

图 3-9 基于状态转移图的测试用例

5. 基于状态的上下文覆盖

基于状态的上下文覆盖类似于继承上下文覆盖。它提供了传统结构化覆盖率度量的一个可选择的定义。这些可选择的定义在不同的上下文内其独立度量的覆盖率不同。

基于状态的上下文覆盖对应于被测类对象的潜在状态。这样的基于状态的上下文覆盖把一个状态上下文内的一个例行程序的执行认为是独立于另一个状态内相同例行程序的执行。为了达到 100% 的基于状态的上下文覆盖，例行程序必须在每个适当的上下文(状态)内被执行。

我们以计算一个类的人口点覆盖率为例。对于一个类来说，在某个状态下的人口点覆盖是其在该状态下被调用的方法的数量除以类中方法的总数。因为一个类存在多个状态，必须在每个状态下计算其人口点覆盖率。因此对该类来说，总的基于状态上下文人口点覆盖率被定义为对应于该类在每个状态中的基于状态上下文人口点覆盖的平均值。具体可以通过下面的公式计算：

$$\text{基于状态的上下文人口点覆盖率} = \frac{\text{累加每个状态内执行到的方法数}}{\text{状态数} \times \text{类内方法总数}}$$

注意，为了使用基于状态的上下文覆盖，构造器不被认为是类的方法。(在新的研究中，为了考虑构造函数和析构函数的测试，增加了两个状态， α 状态和 ω 状态^[127])

3.4.3 基于线程的上下文覆盖

支持继承上下文覆盖和基于状态的上下文覆盖的上下文覆盖概念可以用于帮助进行其他覆盖率分析。

例如，当测试一个多线程程序时，传统的结构化覆盖率度量会集合来自所有线程的覆盖率得到一个单个的覆盖率值。而基于线程的上下文覆盖方法可以应用在这里以维护每个线程的独立的覆盖率信息。

3.5.3 判定路径覆盖

判定路径覆盖(Decision-to-Decision Paths Coverage, DDP Coverage)是判定覆盖的一个变体。这里的判定是指一个序列语句,其起始位置是函数入口或一个判定(如 If, while, switch 等)的开始,结束位置是下一个判定的开始。具体可以见图 3-11。

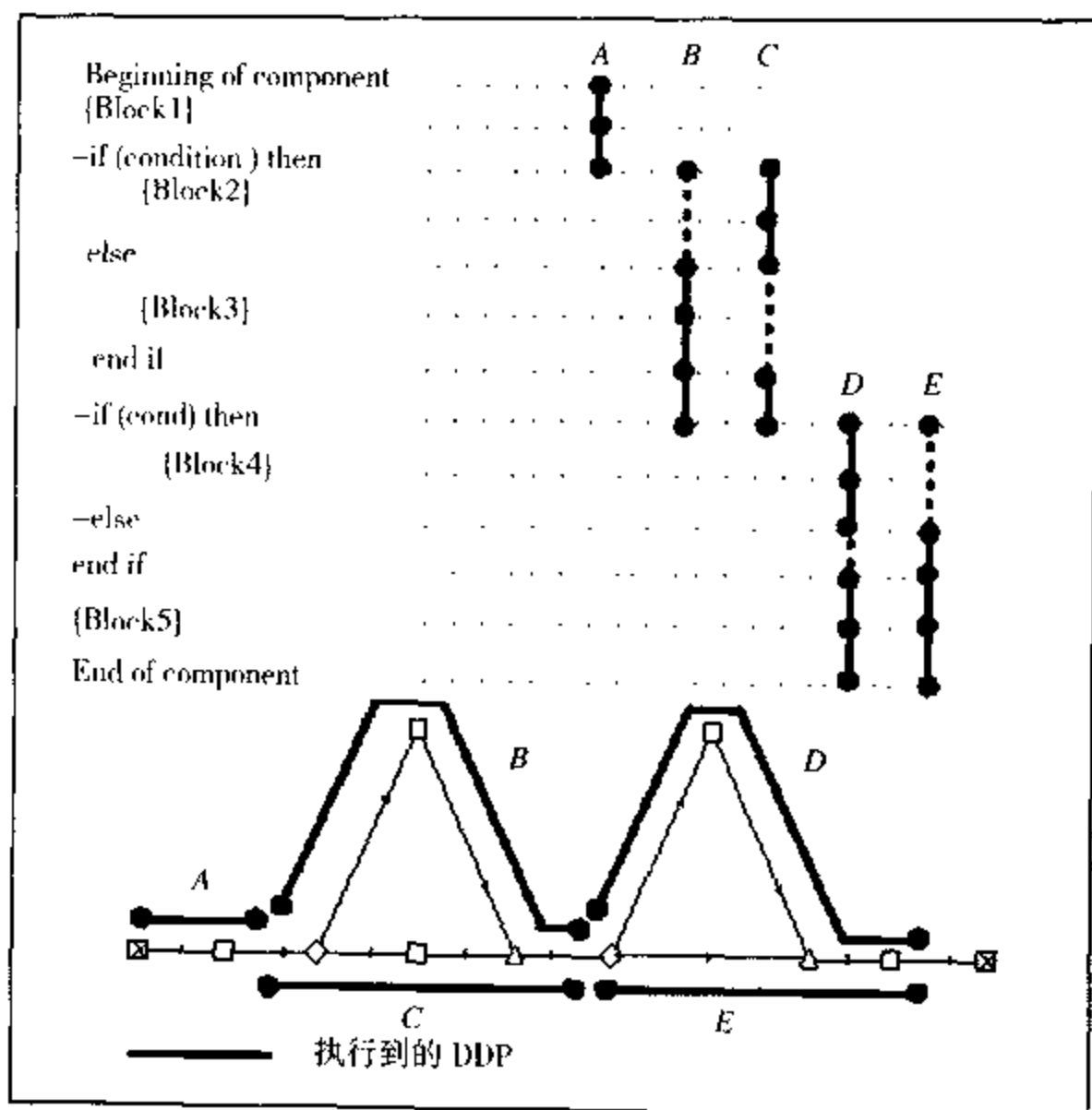


图 3-11 DDP 概念

通过计算哪些判定路径已经走过,哪些没有走过,我们就可以得到 DDP 覆盖率了。DDP 覆盖率公式如下:

$$\text{DDP 覆盖率} = (\text{至少被执行到一次的判定路径数量}) / (\text{系统中判定路径总数})$$

3.5.4 更改条件判定覆盖

更改条件判定覆盖(Modified Conditions/Decisions Coverage, MC/DC Coverage)是判定条件覆盖的一个变体。它主要为多条件测试的情况提供了方便。通过分析条件判定的覆盖来增加测试用例,防止测试呈指数上升趋势。MC/DC 标准满足下列需求:

需求 1 被测试程序模块的每个入口点和出口点都必须至少被走一次。并且每一个程序判定的结果至少被覆盖一次。

需求 2 通过分解逻辑操作,程序的判定被分解为基本的布尔条件表达式,每个条件

独立地作用于判定的结果，覆盖所有条件的可能结果。

图 3-12 是一个简单的 MC/DC 例子。

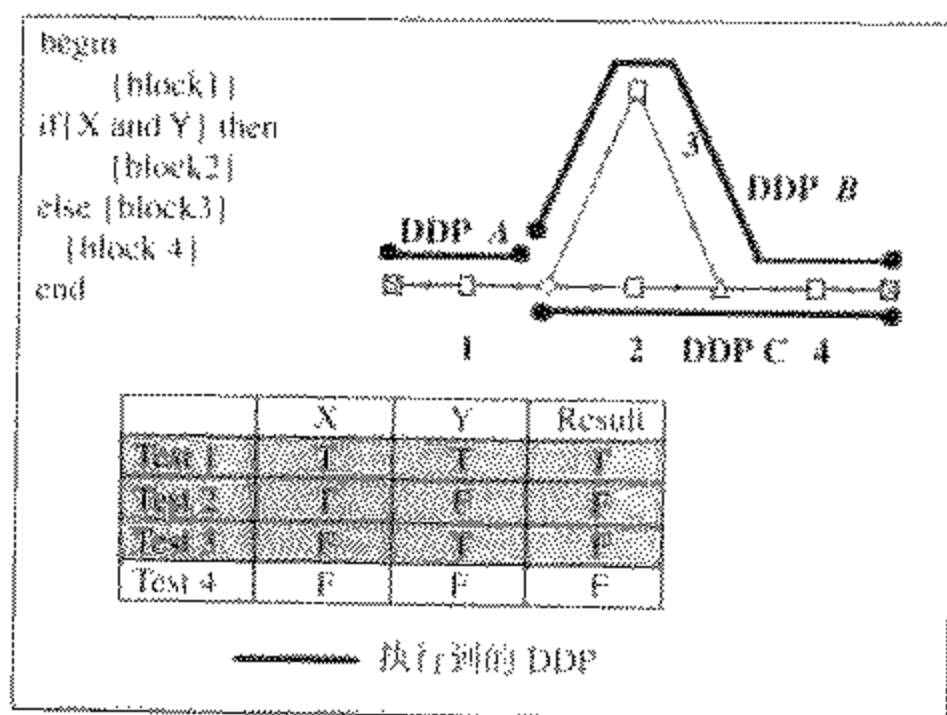


图 3-12 MC/DC 例子

在这个例子中，如果执行 test1 (覆盖 DDPs A 和 C) 和 test3 (覆盖 DDPs A 和 B)，即满足了需求 1，并且达到了 DDP 覆盖 100%。但表达式 Y 其值为 F 的情形没有被考虑到。这意味着需要进行更多的测试才能满足需求 2。

从图 3-12 可知，test1 是必须执行的，因为该用例是唯一使判定结果为 True 的用例。test3 也是必须的，因为该用例使唯一的一个可以通过修改 X 的值改变判定结果的。同样 test2 也是必须的。因此这三个用例满足了需求 1 和需求 2。

再看一个例子：关于判定 (X and (Y or Z)) 的 MC/DC 情况。首先看图 3-13 给出的分析表格。

	X	Y	Z	Result	X	Y	Z
Test 1	T	T		T	5		
Test 2	T	T	F	T	6	4	
Test 3	T	F	T	T	7		4
Test 4	T	F	F	F		2	3
Test 5	F	T	T	F	1		
Test 6	F	T	F	F	2		
Test 7	F	F	T	F	3		
Test 8	F	F	F	F			

图 3-13 MC/DC 计算表格

为了使判断 X 对判定结果独立起作用，假设 Y 和 Z 都是 True，因此 test1 和 test5 是必须的。同样为了使判断 Y 对判定结果独立起作用，需要假定 X 是 True，Z 是 False，因此 test2 和 test4 是必须的。为了使判断 Z 对判定结果独立起作用，需要假定 X 是 True，Y 是

False, 因此 test3 和 test4 是必须的。作为结果测试用例“1, 5, 2, 4, 3”满足 MC/DC 覆盖。很明显这不是惟一可能的组合。

3.5.5 分支条件组合覆盖

分支条件组合覆盖(Branch Condition Combination Coverage)是一种比判定条件覆盖更强的覆盖。它的含义是, 设计一定的测试用例, 使每个分支中的各操作数值的组合都遍历一次。分支条件组合覆盖率的计算公式如下:

分支条件组合覆盖率 = (被评价到的分支条件组合数)/(分支条件组合总数)

看下面这个例子:

```
1. if A and B then
2.   DO_SOMETHING;
3. end if;
```

其中, A 和 B 是操作数, 应用判定条件覆盖的概念, 可以设计如表 3-5 给出的一组用例, 以达到判定条件覆盖要求。

表 3-5 满足判定条件覆盖的用例

用例序号	A 的值	B 的值
Case1	True	True
Case2	False	False

显而易见, 在表 3-5 中, A 和 B 的条件组合有 4 种 (True, True), (True, False), (False, True), (False, False)。因此, 如果要达到分支条件组合覆盖, 还得补充两个用例: (True, False), (False, True)。

3.5.6 过程到过程路径覆盖

过程到过程路径覆盖(Procedure-to-Procedure Path Coverage, PPP Coverage)是针对系统级或子系统级的。通过分析 PPP 覆盖, 可以对集成测试提出指导。一个 PPP 对应的是两个函数之间的一个调用关系, 见图 3-14。PPP 覆盖率的公式如下:

PPP 覆盖率 = (至少被执行到一次的 PPP 数量)/(系统中 PPP 总数)

3.5.7 Z 路径覆盖

Z 路径覆盖是路径覆盖的一个变体。路径覆盖是白盒测试最为典型的问题。着眼于路径分析的测试可称为路径测试。完成路径测试的理想情况是做到路径覆盖。对于比较简单的小程序实现路径覆盖是可能做到的。但是如果程序中出现多个判断和多个循环, 可能的路径数目将会急剧增长, 达到天文数字, 以致实现路径覆盖不可能做到。

为了解决这一问题, 我们必须舍掉一些次要因素, 对循环机制进行简化, 从而极大地

减少路径的数量,使得覆盖这些有限的路径成为可能。我们称简化循环意义下的路径覆盖为 Z 路径覆盖。

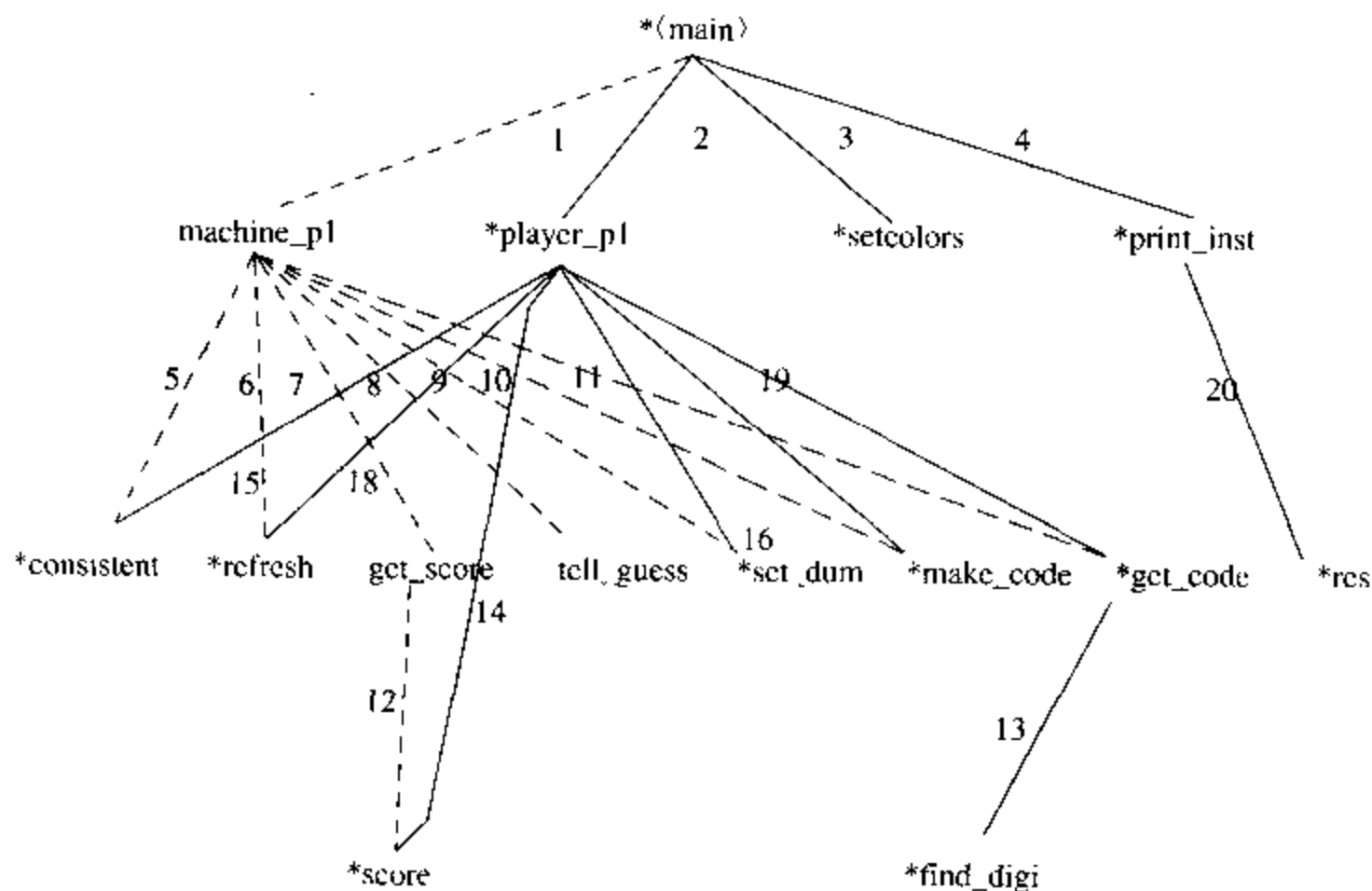


图 3-14 PPP 调用关系

这里所说的对循环的简化是限制循环的次数。无论循环的形式及实际执行循环体的次数多少,只考虑循环一次和零次两种情况。也即只考虑执行时进入循环体一次和跳过循环体这两种情况。图 3-15 中(a)和(b)表示了两种最典型的循环控制结构。前者先作判断,循环体 B 可能执行(假定只执行一次),也可能不执行。这就如同(c)所表示的条件选择结构一样。后者先执行循环体 B(假定也执行一次),再经判断转出,其效果也与(c)中给出的条件选择结构只执行右支的效果一样。

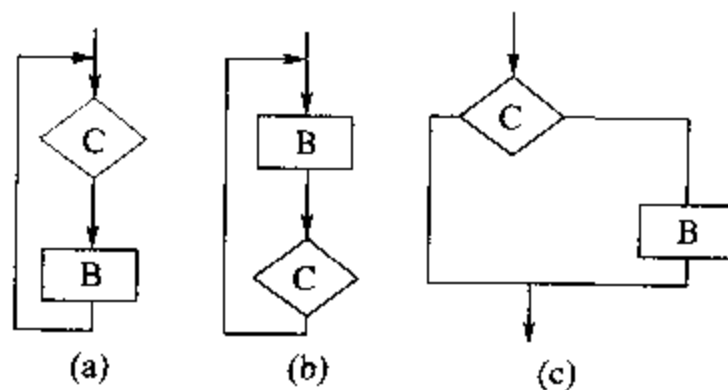


图 3-15 简化循环结构

对于程序中的所有路径可以用路径树来表示,具体表示方法本书略。当得到某一程序的路径树后,从其根结点开始,一次遍历,再回到根结点后,把所经历的叶结点名排列起来,就得到一个路径。如果设法遍历了所有的叶结点,那就得到了所有的路径。

当得到所有的路径后,生成每个路径的测试用例,就可以做到 Z 路径覆盖测试。

3.5.8 ESTCA 覆盖

本章 3.2 节所介绍的逻辑覆盖其出发点似乎是合理的。所谓“覆盖”，就是想要做到全面，而无遗漏。但事实表明，它并不能真正地做到无遗漏。例如，如果把：

$$\begin{cases} \dots \\ \text{IF } (I = 0) \\ \text{THEN } I = J \\ \dots \end{cases}$$

错写成：

$$\begin{cases} \dots \\ \text{IF } (I \neq 0) \\ \text{THEN } I = J \\ \dots \end{cases}$$

如果使用前面的覆盖，我们就无法发现这个问题。

出现这种情况的原因在于，错误区域仅仅在 $I = 0$ 这个点上，即仅当 I 取 0 时，测试才能发现错误。它的确在我们力图全面覆盖来查找错误的测试“网上”钻了空子；并且恰恰在容易发生问题的条件判断那里未被发现。面对这类情况，应该从中吸取的教训是测试工作要有重点，要多针对容易发生问题的地方设计测试用例。

K. A. Foster 从测试工作实践的教训出发，吸收了计算机硬件的测试原理，提出了一种经验型的测试覆盖准则，较好地解决了上述问题^[35]。

经验型覆盖准则是从硬件的早期测试方法中得到启发的。在硬件测试中，对每一个门电路的输入、输出测试都是有额定标准的。通常，电路中一个门的错误常常是“输出总是 0”，或是“输出总是 1”。与硬件测试的这一情况类似，我们常常要重视程序中谓词的取值。但实际上它可能比硬件测试更加复杂。Foster 通过大量的实验确定了程序中谓词最容易出错的部分，得出了一套错误敏感测试用例分析规则 ESTCA (Error Sensitive Test Cases Analysis)。事实上，规则十分简单：

规则 1 对于 $A \text{ rel } B$ (rel 可以是 $<$, $=$ 和 $>$) 型的分支谓词，应适当地选择 A 与 B 的值，使得测试执行到该分支语句时， $A < B$, $A = B$ 和 $A > B$ 的情况分别出现一次。

规则 2 对于 $A \text{ rel1 } C$ (rel1 可以是 $>$ 或是 $<$, A 是变量， C 是常量) 型的分支谓词，当 rel1 为 $<$ 时，应适当地选择 A 的值，使：

$$A = C - M$$

(M 是距 C 最小的容许正数，若 A 和 C 均为整型时， $M = 1$)。同样，当 rel1 为 $>$ 时，应适当地选择 A ，使：

$$A = C + M$$

规则 3 对外部输入变量赋值，使其在每一测试用例中均有不同的值与符号，并与同一组测试用例中其他变量的值与符号不一致。

显然，上述规则 1 是为了检测 rel 的错误，规则 2 是为了检测“差一”之类的错误（如本应是“IF $A > 1$ ”而错成“IF $A > 0$ ”），而规则 3 则是为了检测程序语句中的错误（如应引用一变量而错成引用一常量）。

上述 3 条规则并不是完备的,但在普通程序的测试中却是有效的。原因在于规则本身针对着程序编写人员容易发生的错误,或是围绕着发生错误的频繁区域,从而提高了发现错误的命中率。

试运用这些规则来检验上面的小程序段错误。应用规则 1,对它测试时,应选择 I 的值为 0,使 $I = 0$ 的情况出现一次。这样一来就立即找出了隐藏的错误。

当然,ESTCA 规则也有很多缺陷。一个缺陷是,有时不容易找到合适的输入数据,使规则所指的变量值满足要求。另一个缺陷是,仍有很多错误发现不了。对于查找错误的广度问题在变体测试(Mutation Testing,参考第 5 章相关内容)中得到较好的解决。

3.5.9 LCSAJ 覆盖

LCSAJ 覆盖(Linear Code Sequence and Jump Coverage)是 Woodward 等人提出来的。一套覆盖率准则,意思是线性代码序列与跳转覆盖。一个 LCSAJ 是一组顺序执行的代码,以控制流跳转为其结束点。它的定义如下:

- 它起始于程序的入口或者一个可能导致控制流跳转的点;
- 它结束于程序的出口或者一个可能导致控制流跳转的点;
- 对于该点,一个跳转在后面的序列中产生。

它不同于 DDP。DDP 是根据程序有向图决定的。一个 DDP 是指两个判断之间的路径,但其中不再有判断。程序的入口、出口和分支结点都可以是判断点。而 LCSAJ 的起点是根据程序本身决定的。它的起点是程序第一行或转移语句的入口点,或是控制流可以跳达的点。因此,几个 LCSAJ 首尾相接构成 LCSAJ 串,组成程序的一条路径。第一个 LCSAJ 起点为程序起点,最后一个 LCSAJ 终点为程序终点。一条程序路径可能是由两个、三个或多个 LCSAJ 组成的。基于 LCSAJ 与路径的这一关系,Woodward 提出了 LCSAJ 覆盖准则。这是一个分层的覆盖准则:

[第一层]:语句覆盖。

[第二层]:分支覆盖。

[第三层]:LCSAJ 覆盖。即程序中的每一个 LCSAJ 都至少在测试中经历过一次。

[第四层]:两两 LCSAJ 覆盖。即程序中每两个首尾相连的 LCSAJ 组合起来在测试中都要经历一次。

...

[第 $n+2$ 层]:每 n 个首尾相连的 LCSAJ 组合在测试中都要经历一次。

这说明了,越是高层的覆盖准则越难满足。

在实施测试时,若要实现上述的 Woodward 层次 LCSAJ 覆盖,需要产生被测程序的所有 LCSAJ。

尽管 LCSAJ 覆盖要比判定覆盖复杂得多,但是 LCSAJ 的自动化相对还是容易获得的,例如 IPL 的 Cantata 系列工具可以提供这一覆盖。

对一个模块的微小变动可能对 LCSAJ 产生重大影响,因此维护 LCSAJ 的测试数据是相当困难的。一个大模块包含极其庞大的 LCSAJ,因此要获得 100% 的覆盖率也是不现实的。然而 Woodward 等人提供的证据表明把测试 100% 的 LCSAJ 作为目标比 100% 的判定覆

盖要有效得多。

3.6 如何使用覆盖率

3.6.1 基本原则

本章给出了许多覆盖率概念。但是，在实际测试中，还会遇到其他一些覆盖率的概念。面对如此众多的覆盖率概念令测试人员感到无所适从。那么该怎么去选择覆盖率，并如何正确使用覆盖率呢？笔者认为，在选择和使用覆盖率时，必须基于以下几个原则：

- 覆盖率不是目的，只是一种手段

测试的目标是尽可能去发现错误，寻找被测对象与既定规格不一致的地方。因此，在进行测试用例设计时，首先应当从对需求和设计的了解，使用已有的经验去挖掘测试用例，包括正常的用例和异常的用例。在这个基础上，再使用需要的覆盖率准则来衡量已有的测试设计，并补充相应的用例来达到需要的覆盖率准则。如果仅以覆盖率目标来指导测试，就会丢失很多重要的信息，陷入追求覆盖率数字的极端现象中去。

- 你不可能针对所有的覆盖率指标去进行测试；相反，只考虑一种覆盖率指标也是不恰当的

如前所说，覆盖率的种类很多，有简单的也有复杂的。如果要针对每种覆盖率都进行测试，就会使测试成本变得十分昂贵，测试本身也将变得无法实现。现在的软件开发讲究市场的快速适应，这就要求我们必须使用尽可能短的，能够让产品迅速稳定下来的测试。所以，对测试就提出了更高的要求。一般产品都希望能够花最小的代价得到最大的测试效果。同样，如果只考虑一种覆盖率指标，会使得测试过程遗漏一些重要的方面。不同的覆盖率所考虑的重点不同，尤其当我们只考虑比较弱的覆盖率时，如语句覆盖，就会丢失很多在路径、判定、条件上的错误信息。

- 不要追求绝对 100% 的覆盖率

当然，如果有足够的资源和时间，达到 100% 的覆盖是现实的。但事实上，为了要达到 100% 的覆盖率将付出极大的成本，也不是必要的。对于一般性的软件（对于涉及到生命安全方面的系统要求有 100% 的覆盖率）应当设置一个合理的覆盖率标准。研究表明，80% 的错误隐藏在 20% 的代码中。从成本效率考虑，应当把覆盖率和代码静态分析工具结合在一起使用，去覆盖那些最容易出错的代码，而忽略那些简单的和不易出错的代码。

另一个方面，在设计用例时，尽可能地设计高覆盖率的用例。但这些用例是否一定要通过代码执行来验证却是可以变通的。笔者曾经遇到一些测试人员，他们为了使覆盖率工具能够显示 100% 的覆盖率，花很多时间去模拟一些异常情况的桩模块。然而这些异常情况在系统正常运行中其实是很少发生的。如为了验证函数中对内存分配不成功做出的处理就是一个例子。例如在下面的代码中：

```
char *pBuf = NULL;
...
pBuf = (char *) malloc(100);
```

```

if (pBuf == NULL)
{
    printf("Not Enough Memory\n");
    return -1;
}

```

测试人员对 malloc 进行了打桩，使其返回一个 NULL。这样，例中最后两行代码就可以执行到了，也能达到 100% 的覆盖率。但是，可以思考一下，是否必须这么做呢？就像原则 1 中提到的，测试不应是为了覆盖率而使用覆盖率。其实，对于此种情况，尽管在设计用例时必须考虑这种异常的用例，而在执行时，完全可以通过人工走读的方式来验证这些用例，这将会灵活得多，成本也会降低。毕竟写桩模块的成本是很高的。

3.6.2 一个选择建议

在这个建议中，笔者使用 5 个维度来衡量不同的覆盖率标准。它们分别是：

- 可自动化性
- 可获得性
- 可理解性
- 可维护性
- 完整性

对于每个维度，又给出 5 种不同的级别：好，较好，一般，较差，差。表 3-6 给出了一些主要覆盖率的评价。

表 3-6 覆盖率评价

覆盖率度量	评价标准				
	可自动化性	可获得性	可理解性	可维护性	完整性
语句覆盖	好	好	好	好	差
判定覆盖	好	好	好	好	较差
条件覆盖	较好	好	好	好	一般
判定条件覆盖	较好	好	好	好	较好
路径覆盖	差	较差	较差	较差	好
IB 覆盖	好	好	好	好	差
DDR 覆盖	好	好	好	好	较差
MC/DC 覆盖	较好	好	好	好	较好
分支条件组合覆盖	较好	较好	较好	好	较好
ESTCA 覆盖	较差	较差	一般	较好	一般

第1个分支谓词引出的两个操作，及第2个分支谓词引出的两个操作组合起来而得到的，即 $2 \times 2 = 4$ 。这里的“2”是由于两个并列的操作， $1 + 1 = 2$ 而得到的。

对于一般的、更为复杂的问题，估算最少测试用例数的原则也是相同的。现以图 3-18 表示的程序为例。该程序中共有 9 个分支谓词，尽管这些分支结构交错起来似乎十分复杂，很难一眼看出至少应需要多少个测试用例。但如果仍用上面的方法，也是很容易解决的。首先应注意到，该图可分上下两层：分支谓词 1 的操作域是上层，分支谓词 8 的操作域是下层。这两层正像前面简单例中的 P1 和 P2 的关系一样。只要分别得到两层的测试用例个数，再将其相乘即得总的测试用例数。这里，需要首先考虑较为复杂的上层结构。谓词 1 不满足时的操作又可进一步分解为两层，就是图 3-19 中的 (a) 和 (b)。它们所需测试用例个数分别为 $1 + 1 + 1 + 1 + 1 = 5$ 及 $1 + 1 + 1 = 3$ 。因而两层组合，得到 $5 \times 3 = 15$ 。于是整个程序结构的上层所需测试用例数为 $1 + 15 = 16$ ，而下层显然为 3。故最后得到整个程序所需测试用例数至少为 $16 \times 3 = 48$ 。

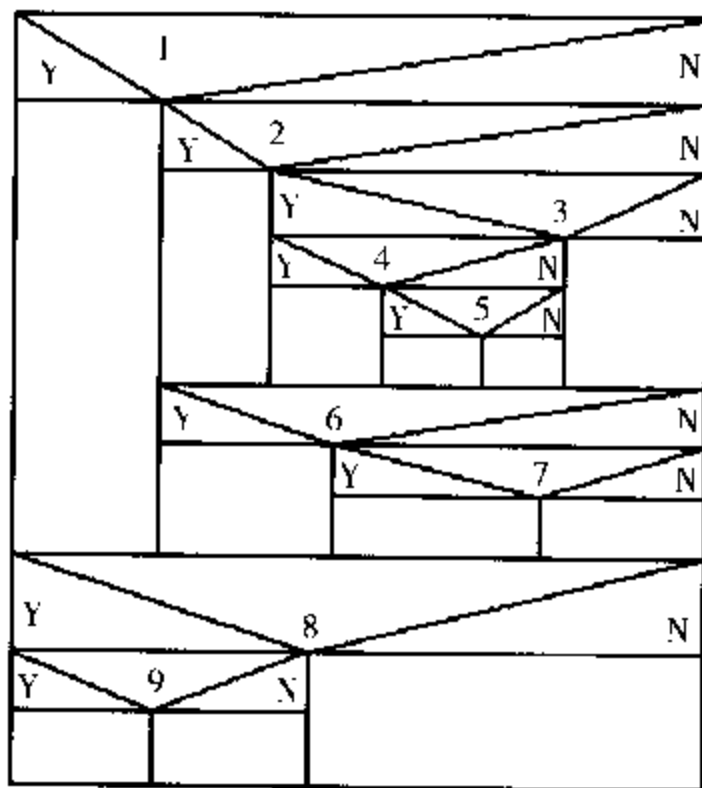


图 3-18 计算最少测试用例数

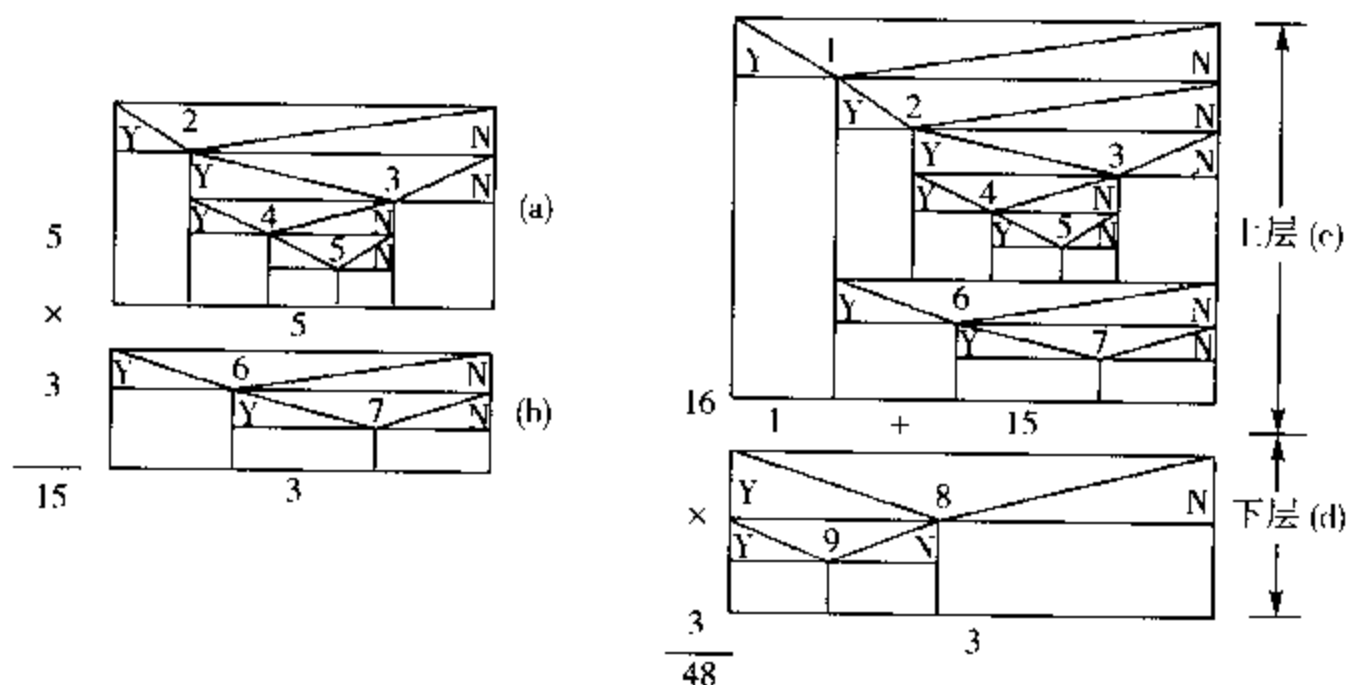


图 3-19 最少测试用例数计算

3.7 本章小结

本章介绍了经常用到的一些覆盖率概念，其中 3.2 和 3.3 节讲述的是最基本的覆盖率概念。对于那些从事测试工作的人来说，需要掌握 3.4 和 3.5 节中的内容。覆盖率是一种度量测试完整性的指标，但覆盖率不是测试的目标，只是一种手段。

第4章 程序分析技术

程序分析技术是软件测试的一个基本技能，只有掌握了这项技能才能真正有效地设计测试用例。在本章，主要对于一些能够支持测试分析的比较成熟的程序分析技术进行一个概要的阐述，包括基本概念，简单的例子讲解等。通过本章的学习，你需要掌握：

1. 常见的程序分析方法有哪些？
2. 什么是从文本视角进行分析？
3. 程序插装是从哪个视角对程序进行分析的？
4. 程序插装可以应用到哪些领域？
5. 程序插装和故障插入之间有什么联系和区别？
6. 控制流分析和数据流分析之间有些什么联系和区别？
7. 什么是程序切片技术？
8. 故障插入技术有什么用？其难点是什么？
9. 什么是变体分析？如何进行变体操作？

在实践中，我们已经认识到，每个测试技术需要确定软件特性；而发现这些特性的技术称为分析。由于一个测试技术可能需要依赖多个分析技术，同时一个分析技术又可能支持多个测试技术，因此分析技术被独立于它们支持的测试技术而进行讨论。

任何探索确定软件特性的技术都是程序分析的一种形式。软件特性在开发、调试、文档化、验证、评价、证明和维护中是很关键的。本章主要描述支持验证和测试的分析技术。此外，分析可以帮助决定什么地方需要集中测试。

在所有测试阶段使用到的分析包括：测试数据选择、程序执行、数据收集和结果评价等几个方面。测试数据的选择可以基于不同信息源的考虑，包括规格、实现和潜在的错误及故障。收集计算信息可能需要对程序文本进行分析并进行程序插装。程序执行本身就是一种分析形式。建立一个用于验证的准则可能需要进行额外的分析。

这里讨论的分析技术从程序的不同视角进行了分类。每一种视角强调程序的不同特性，并且能够确定不同程序的特点。一些分析技术可能需要使用多种不同视角。下面根据视角包含信息的递增次序来讨论这些视角。

4.1 文本视角

从文本的视角(a Textual View)来看，一个程序被看作是字符或记号的序列。许多主要的度量，例如程序长度和标识符频度，使用了这种视角。赫尔斯梯德度量是这种视角的基本应用^[106]。文本编辑器把程序作为文本对象进行处理，同时做文本扫描和行计算等。例如，常用的 Word、UltraEdit、Windows 自带的记事本都是这方面的例子。编程规范(用

于程序编写的格式规范或约定)经常是从这个观点进行表达的。

文本视角提供的信息是物理的、基本的。它一般不对文本内容的语法和语义进行分析,而只对文本本身的基本内容进行分析。例如,分析文本的规模、行数、词汇数、词汇频度等内容。文本视角是其他视角分析的基础。

4.2 句法视角

一个程序可以被看成一个分层结构。它可以被分解到子程序,再被分解到语句组,然后被分解到语句等等,直到符号层。在这个分解中,每个最基本的元素以及元素间的组合关系是由程序语言的语法决定的。这种视角被称为句法视角(a Syntactic View)。句法视角可以从一个文本视角来获得(例如,通过一个语法分析器),或者可以被直接构成(例如通过一个直接句法编辑器)。可派生的程序特性包括语句计数、标识符交叉引用、程序调用图、声明和未声明的变量、变量使用的频度等等。许多成熟的程序度量就是基于这种视角的。

句法视角支持程序插装技术,其中通过修改源代码或者目标代码来揭示其内部的工作原理。在程序执行过程中,许多程序特性可以被确定。例如,哪些语句或分支被执行到。可以获得执行语句的数量或者完全跟踪每个表达式被计算的值。Probert 提出了对这种插装技术的算法^[49],Beizer 研究了关于插装的级别以及它们导致的影响^[26]。插装过的代码必然比原来的代码要大且运行得慢。因此,执行插装代码可能会掩盖时间、规模和位置等相关的问题。尤其对于实时系统,插装过的代码可能会导致时序的不一致,而这可能不是程序的问题而是因为插装本身带来的问题。

程序插装

简单地说,程序插装方法是借助向被测程序中插入操作来实现测试目的的方法。

在调试程序时,常常要在程序中插入一些打印语句。其目的在于,希望执行程序时,打印出我们最为关心的信息。通过这些信息进一步了解执行过程中程序的一些动态特性。例如,程序的实际执行路径,特定变量在特定时刻的取值等。从这一思想发展出的程序插装技术能够按用户的要求,获取程序的各种信息,成为测试工作的有效手段。

可能的程序插装应用包括:

- 测试覆盖率和测试用例有效性度量;
- 断言检测;
- 数据流异常检测;
- 路径智能分解(Pathwise Decomposition)。

1. 用于测试覆盖率和测试用例有效性度量的程序插装

可能的测试覆盖率度量包括,但不限于下面三种类型(请参考本书第3章的内容):

- 程序中每个语句至少被执行一次;

- 在流程图中的每个分支至少被经过一次；
- 每个可能被执行的路径至少被经过一次。

那么，怎样才能做到呢？可以通过以下途径：

- 在控制流中确定一个点集。这样，如果我们知道每个点在执行期间被经过的次数，就可以确定每个语句被执行的次数了（或者，每个分支被经过的次数）；
- 在程序中的这些点上插入软件计数器；
- 用一组测试用例测试这个程序；
- 检查计数器的值以确定获得的覆盖率的范围。

那么，如何建立一个软件计数器呢？计数器的实现可以使用一个已命名的子程序或子过程。假设， $\text{count}(j)$ 。这个子程序使用一个整数数组 $\text{counter}[1..n]$ 。数组中的每个元素被初始化为 0。一次 $\text{count}(j)$ 的调用会引起相应数组 $\text{counter}[j]$ 加 1。

以“求整数 X 和 Y 的最大公约数”的程序为例，说明该程序插装方法的要点。图 4-1 给出了这个程序的流程图。图中虚线框不是原程序的代码，而是为了记录语句执行次数而插入的代码。它们要执行的操作都是计数语句（这里没有使用子程序，当然也可以定义一个子程序 $\text{count}(j)$ 代替该语句，效果相同），其形式为 $C(i) = C(i) + 1 (i = 1, 2, \dots, 6)$ 。

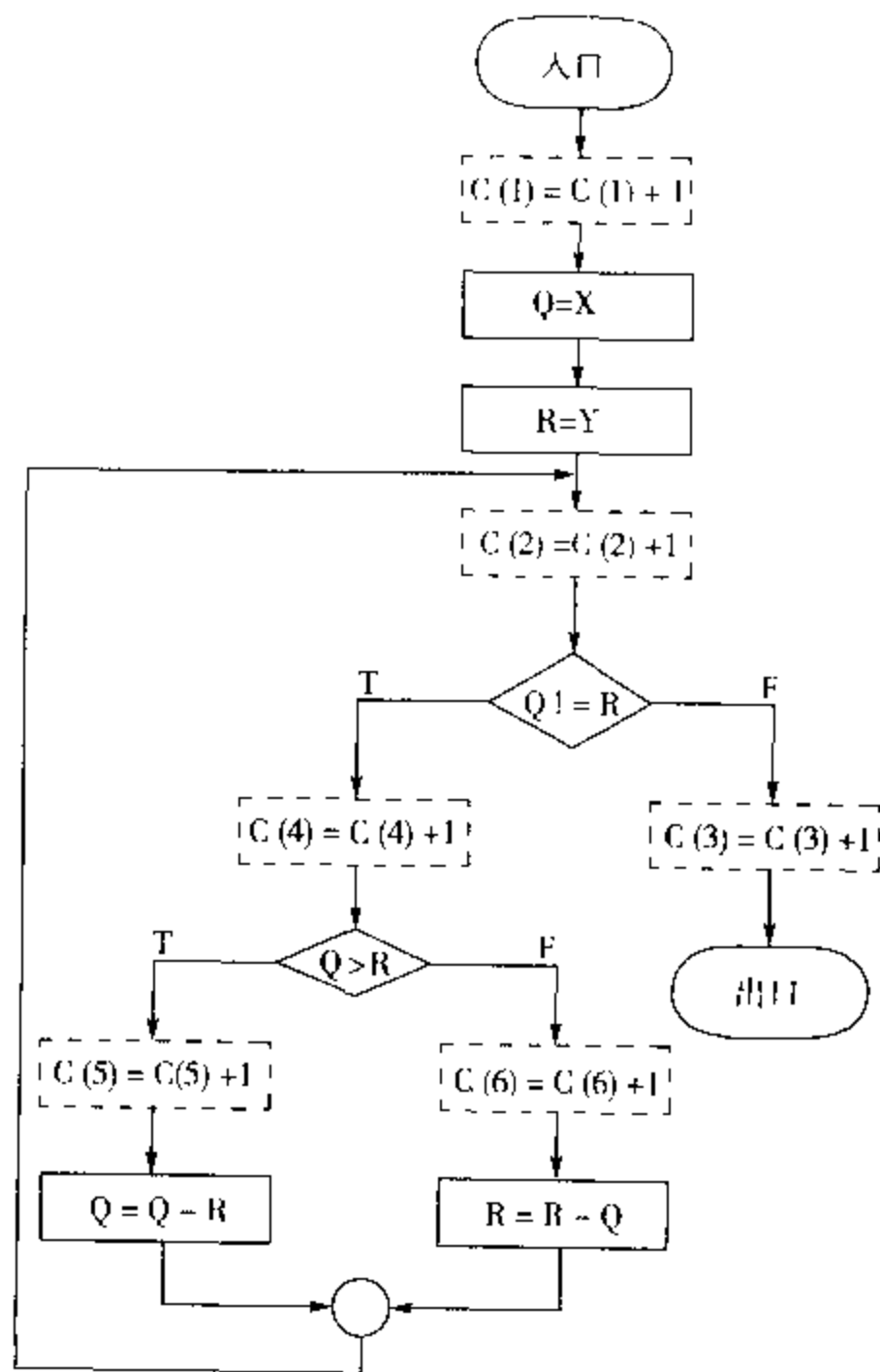


图 4-1 求最大公约数插装后流程图

程序从入口开始执行,到出口结束。凡经历的计数语句都能记录下该程序点的执行次数。如果在程序的入口处还插入了计数器 $C(i)$ 初始化的语句,在出口处插入了打印计数器的语句,就构成了完整的插装程序。它便能记录并输出各程序点上语句的实际执行次数。

设计插装程序时需要考虑的问题包括:

- 探测哪些信息;
- 在程序的什么部位设置探测点;
- 需要设置多少个探测点。

其中,前两个问题需要结合具体情况解决,并不能给出笼统的回答。至于第三个问题,需要考虑如何设置最少探测点的方案。一般情况下,在没有分支的程序段中只需一个计数语句。若程序中出现多种控制结构,使整个结构十分复杂时,为了在程序中设计最少的计数语句,需要针对程序的控制结构进行具体的分析。这里以 FORTRAN 程序为例,列举至少应在哪些部位设置计数语句的建议:

- 程序块的第一个可执行语句之前;
- ENTRY 语句的前后;
- 有标号的可执行语句处;
- DO、DO WHILE、DO UNTIL 及 DO 终端语句之后;
- BLOCK-IF、ELSE IF、ELSE 及 ENDIF 语句之后;
- LOGICAL IF 语句处;
- 输入/输出语句之后;
- CALL 语句之后;
- 计算 GO TO 语句之后。

一个测试用例的有效性,是指该用例揭示程序错误的能力。如果一个测试用例有时会导致程序产生正确的结果,甚至在程序错误的情况下也是如此,那么该用例是无效的。

一个程序为什么会产生偶然正确的结果呢?其中一个原因就是因为它包含类似 exp1 op exp2 格式的表达式,并且使用的测试用例使得 exp1 为这样一个特殊值,它使 $\text{exp1 op exp2} = \text{exp1}$ 而不管 exp2 是什么值。如果在 exp2 中存在错误,那么在测试结果中,这个错误永远不会被发现。表 4-1 是类似表达式和类似测试用例的一些例子。

表 4-1 可能产生偶然正确结果的例子

表达式	测试用例
$(a+b) \times (c-d)$	用例使得 $(a+b) = 0$
$P(x)$ and $Q(y, z)$	用例使得 $P(x)$ 为假
$P(x, y)$ or $Q(z)$	用例使得 $P(x, y)$ 为真

一个程序的一个测试用例的单一指数(Singularity Index)被定义为:在一次特定测试执行期间,一个测试用例惟一地关注于遇到的一个多重表达式的次数。为了自动计算一个测试用例的单一指数,对程序中包含的每个表达式进行一个完整检查和分析是必要的。如果限制在具有 exp1 op exp2 格式的多重表达式上,插装工具可以被设计用来插装多重表达式中的每个基本因子,这样来计算测试用例惟一关注这个因子的次数。

例如,假定程序中有一个语句包含如下表达式: $a \times (c + (d / e))$ 。这个表达式存在 3 个需要关注的因子,分别是: a , $(c + (d / e))$ 和 d 。一个工具可以设计用来插装该表达式,显示如下:

```
if (a == 0) si ++;  
if ((c + (d/e)) == 0) si ++;  
if (d == 0) si ++;  
  
a * (c + (d/e));
```

其中 si 是一个变量,用于存储测试用例的单一指数。这个单一指数越高,测试用例的有效性越低。

2. 用于断言检测的程序插装

在程序的特定部位插入某些用以判断变量特性的语句,使这些语句在程序执行中得以证实,从而使程序的运行特性得以证实。我们把插入的这些语句称为断言(Assertions)。这一作法是程序正确性证明的初等步骤,尽管算不上严格的证明,但方法本身仍然是很实用的。断言是程序插装技术的一个典型使用。通过使用断言检查,可以发现程序中潜在的错误。但是,程序员发现它在实际中不是那么好用。为了有效,程序员必须在程序的正确位置放入正确的断言,这不是一件容易的事情。在应用中寻找正确断言的问题就像证明程序正确性一样。在这方面还没有什么有效的过程。如果使用断言,那么当一个违例产生时,用户得去分析到底是程序错误引起的还是因为不正确的断言引起的。这常常是拒绝断言检测的一个主要原因。图 4-2 是一个断言的例子。

<pre>unsigned int Dev(int num1, int num2, int&ref { int val1, val2, k = 0; val1 = num1; val2 = num2; while((val1 - val2) > 0) { val1 = val1 - val2; k ++; } ref = val1; return k; }</pre>	<pre>unsigned int Dev(int num1, int num2, int&ref) { int val1, val2, k = 0; val1 = num1; val2 = num2; _ASSERT(num1 >= 0); _ASSERT(num2 > 0); while((val1 - val2) > 0); { val1 = val1 - val2; k ++; _ASSERT((num2 * k + val1) == num1); } ref = val1; _ASSERT(ref < num2); return k; }</pre>
---	---

图 4-2 一个断言的例子

3. 用于数据流异常检测的程序插装

关于数据流分析方面的概念可以参考下面 4.4 节的内容。有许多检测数据流异常的方法

法^{[51][107][108]}。这里介绍一种使用程序插装来检测数据流异常的方法^[40]。为此,我们首先假设一个变量在程序执行过程中有4种状态: U 状态(未定义状态)、 D 状态(定义但未被引用状态)、 R 状态(定义且被引用状态)和 A 状态(异常状态),见图4-3。

为了错误检测的目的,假设一个变量在其被显式或隐式声明时处在 U 状态。若要对这个变量进行“定义”动作,那么就进入到 D 状态。这个变量的状态变换参考图4-3,其中 u 表示取消定义动作, r 表示引用动作, d 表示定义动作。从图4-3中可以看出,会产生异常状态的情况是在 U 状态时进行引用操作或者在 D 状态时进行定义或取消定义操作。显然,不需要计算一个变量沿着整个执行路径所有采取的动作顺序,只需要知道序列是否包含 Ur (未定义状态采取引用操作), Dd (已定义未使用状态采取定义操作), Du (已定义未使用状态采取取消定义操作)。由此,可以通过插装监视这些状态。

那么,该怎么做呢?假设要考虑变量 x 在语句 S 处的状态变化。令在进入 S 之前 x 的状态是 q , 在语句 S 处要对 x 进行了 α 操作(α 是 u, d, r 中的一种), 执行完 S 之后 x 的状态是 q' , 定义一个状态转换函数 f , 使得 $q' = f(q, \alpha)$ 。根据图4-3, 可得:

$$\begin{aligned} f(U, d) &= D, & f(U, r) &= A, & f(U, u) &= U; \\ f(D, d) &= A, & f(D, r) &= R, & f(D, u) &= A; \\ f(R, d) &= D, & f(R, r) &= R, & f(R, u) &= U; \\ f(A, d) &= A, & f(A, r) &= A, & f(A, u) &= A; \end{aligned}$$

对于操作 $\alpha\beta$, 有:

$$f(q, \alpha\beta) = f(f(q, \alpha), \beta)$$

为了说明这类插装的特点,现举个例子。图4-4是一个程序路径,例中要对变量 x 进行数据流分析。很明显,程序中的 x 被定义了又定义。若使用上面介绍的插装方法对该程

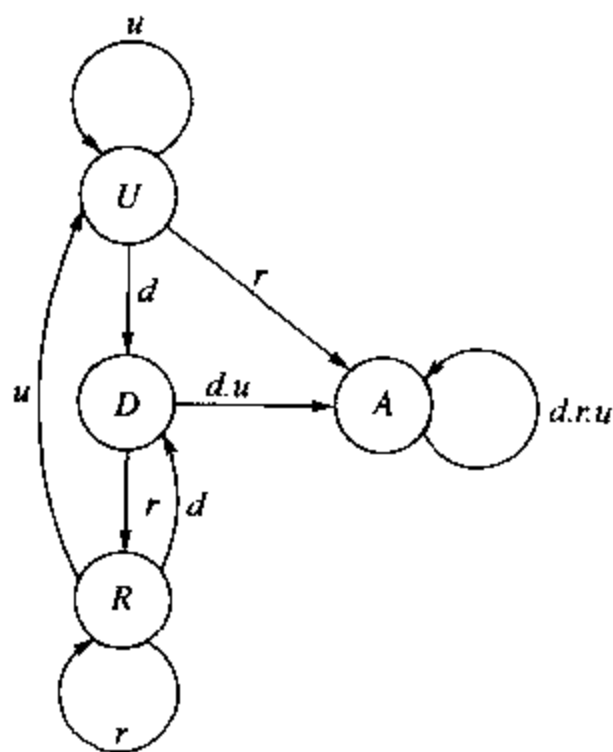


图 4-3 变量在程序中的 4 种状态

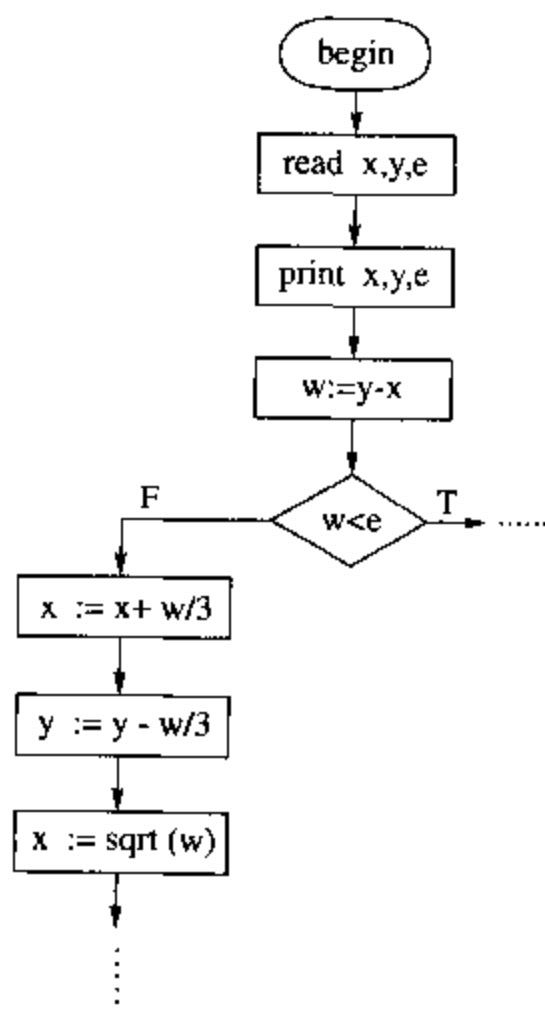


图 4-4 一个程序路径

序插装，则如图 4-5 所示。其中， x 最后的一次状态出现了异常。

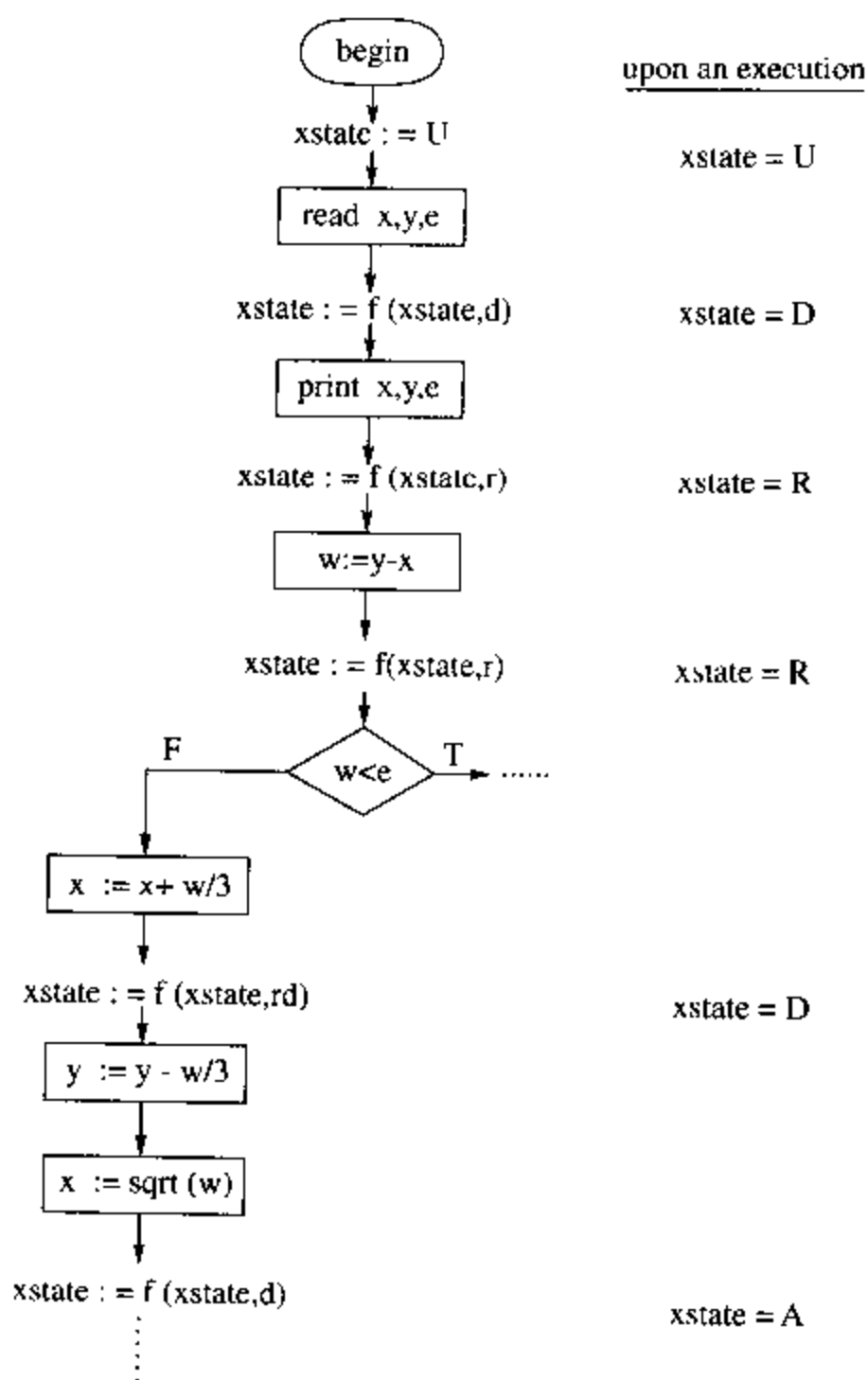


图 4-5 针对变量 x 进行插装后的数据流图

4.3 控制流视角

一个程序的控制流关系 (Control Flow Relation) 叙述了程序元素和它们的执行次序之间的联系。一个程序元素通常是一个条件、一个简单的语句，或者一块语句 (多个连续语句)。如果元素 B 可以在元素 A 之后立即执行，那么 (A, B) 就在程序的控制流关系中。后继的 B 的执行独立于 A 的执行。例如，若 A 指的是条件 $(x < x)$ ， B 指的是写语句 $(write(x))$ ，即：

```
If x < x then write(x)
```

尽管事实上 A 总是为假，但 (A, B) 还是在控制流关系中存在。

对应于控制流关系的图被称为控制流图 (a Control Flow)^[50]。图上的每个节点对应于一个程序元素；两个节点间一个直接的弧表示相应的两个元素在控制流关系中组成一个顺

序对。控制流图的一条路径对应于一个潜在可执行的程序元素顺序。执行一条路径就是执行对应的程序元素序列。如果一个输入引起一条路径的执行,那么这条路径就被称之为是可达的,否则是不可达的。通常,含循环的路径有无限多条路径;即使没有循环,一个程序也可能有非常多的路径需要分析。图4-6是一个控制流图的例子。

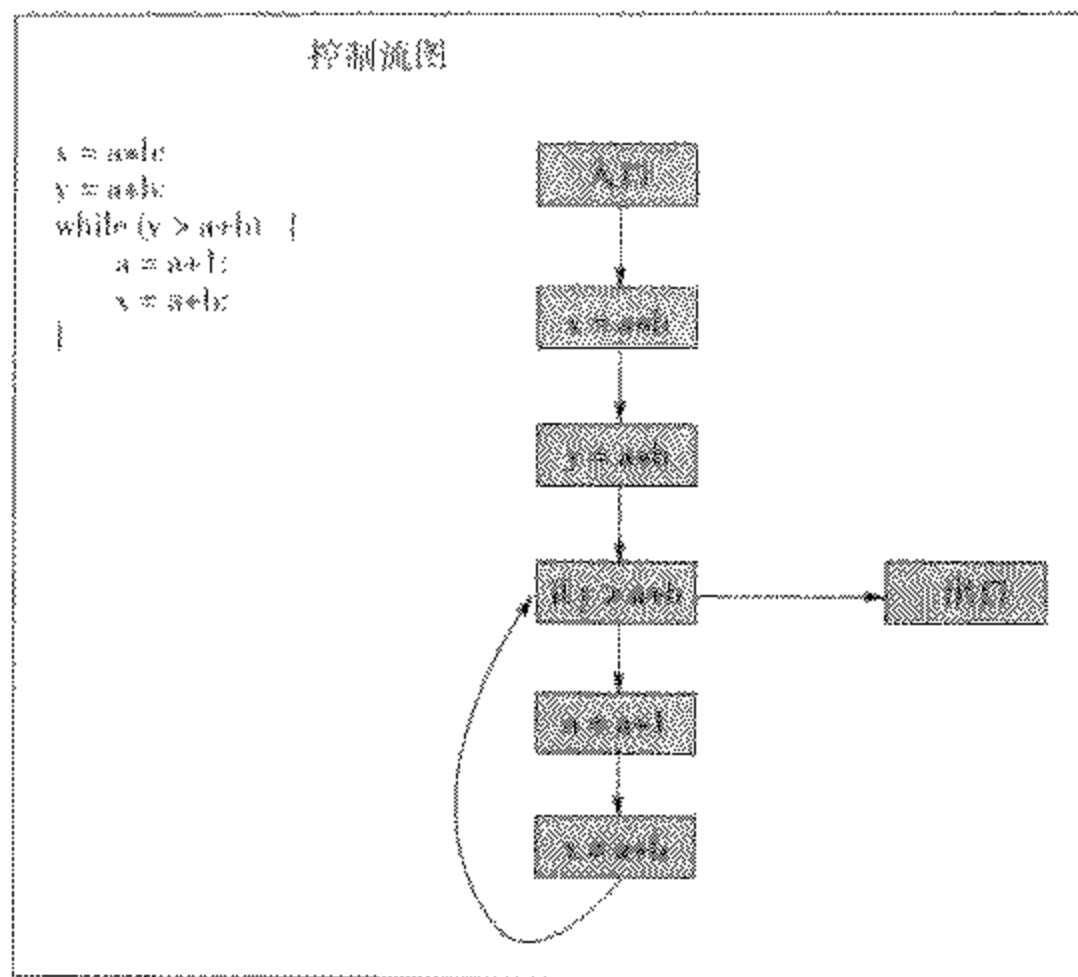


图4-6 控制流图例子

控制流图一般产生于一个句法视角,用于控制流信息的更有效处理。有许多程序度量基于程序的控制流视角。

4.4 数据流视角

程序和程序的元素通过它们的数据访问行为关联起来,这就确定了数据流关系(Data Flow Relation)。如果元素 B 使用一个数据对象,该数据对象潜在地在元素 A 处被定义,那么 (A, B) 就出现在程序的数据流关系中。数据流图(Data Flow Graph)是一个直接被标记的图形。它对应于数据流关系,其中节点对应于程序元素,直接连接 A 到 B 的弧用 v 标记。这里的 v 表示 (A, B) 在数据流关系中是由于在 A 处定义了 v ,在 B 处使用了 $v^{[30][51]}$ 。数据流图可以从句法视角产生的,用于数据流信息的更有效处理。

一个程序可以表示成一个数据流图,使用变量定义、引用和未定义信息做标记。如果一个语句的执行分配了一个值给一个变量,我们称这个变量被定义了。一个变量的引用是指,如果一个语句的执行需要从内存中获得某个变量的值。观察下面的语句:

$x := x * y - z$

其中 y 和 z 都是被引用,而 x 是先被引用,然后又被定义。一个变量在许多情况下可能会

变成未定义。例如，在 FORTRAN 程序中，一个 DO 语句的指示标量在循环结束时就变成未定义的，局部变量在子程序用 RETURN 返回后变成未定义的。

关于数据流的信息可以被引用到代码优化、异常检测和测试数据生成中。

数据流异常值得进行进一步研究，因为它可以揭示一些缺陷。例如，一个变量被定义了两次却没有被使用过，使用一个没有被定义的变量，取消一个自上次定义后就没有被使用过的变量的定义等。检测这些异常的算法在附录参考书目的^[51]和^[53]中提到，并在^[54]中被优化和纠正。具体可以参考 5.2.3 节内容。

程序切片

一个程序切片(slice)是通过在一个特定位置消除那些不影响表达式计算的所有语句产生的^[55]。Korel 使用了切片技术来测试和调试程序。他的方法是使用一个称之为程序依赖图的数据流图上的变量^{[56][57]}。

对于给定的一个程序行为的子集，通过切片技术把程序减小到一个最小化形式，并且仍旧能产生给定的行为。这个简化后的程序就称为切片。看下面的程序：

```
1  begin
2    read(x,y);
3    total := 0.0;
4    sum := 0.0;
5    if x <= 1
6      then sum := y
7      else begin
8          read(z);
9          total := x * y
10         end;
11    write(total, sum)
12  end.
```

第一个切片是要得到变量 z 的值在语句 12 的切片。分析程序可以发现，影响 z 值的语句有 2 和 5，因此得到切片 S1 如下：

```
1  begin
2    read (x, y);
5    if x <= 1
6      then
7      else begin
8          read(z);
10         end;
12  end.
```

同样，若要得到 total 值在语句 12 的切片 S2，则只需考虑如下程序：

```
1  begin
2    read(x,y);
3    total := 0.0;
5    if x <= 1
6      then
```

```

7      else begin
9          tatal := x * y
10         end;
12     end.

```

获得程序行为一个子集的规格称为一个切片标准。它包含程序中的一个特定语句和一个变量的集合。一个切片可以定义如下^[40]：

定义1 P 是一个程序，并且假定其语句被连续标号。对于 P 中的每个语句 n ，我们可以定义两个集合： $REF(n)$ 是所有在语句 n 处被引用的变量的集合； $DEF(n)$ 是在语句 n 处所有被定义的变量的集合。

定义2 程序 P 的一个值的轨迹是一个有限的有序对集合：

$$(n_1, v_1)(n_2, v_2) \cdots (n_k, v_k)$$

其中每个 n_i 表示 P 中的一个语句，每个 v_i 表示在 P 中语句 n 执行前所有变量值的一个向量。

对于上面的例子，变量的一个向量可以表示为 $\langle x, y, z, sum, total \rangle$ ，一个可能的值的轨迹是：

```

T1:  (1, <?, ?, ?, ?, ?>)
      (2, <?, ?, ?, ?, ?>)
      (3, <X, Y, ?, ?, ?>)
      (4, <X, Y, ?, ?, 0.0>)
      (5, <X, Y, ?, 0.0, 0.0>)
      (6, <X, Y, ?, 0.0, 0.0>)
      (11, <X, Y, ?, Y, 0.0>)
      (12, <X, Y, ?, Y, 0.0>)

```

其中，? 表示没有定义。

定义3 程序 P 的一个切片标准是一个有序对 (i, V) ，其中， i 是 P 中的一个语句， V 是 P 中变量的一个子集。

在上例中的两个切片标准可以表示为 $C1 = (12, \{z\})$ 和 $C2 = (12, \{total\})$ 。

定义4 给定一个切片标准 $C = (i, V)$ 和一个值轨迹 T ，我们可以定义一个映射函数 $Proj(C, T)$ ，该函数删除了在值轨迹中，左边成分不是 i 的所有有序对，并且在剩余的有序对中删除了其右边向量中不在 V 中的成分。例如，对于 $Proj((12, \{z\}), T_1)$ ，有：

$$\begin{aligned}
 Proj(C_1, T_1) &= Proj((12, \{z\}), T_1) \\
 &= \{ \cancel{(1, \langle ?, ?, ?, ?, ? \rangle)}, \cancel{(2, \langle ?, ?, ?, ?, ? \rangle)}, \cancel{(3, \langle X, Y, ?, ?, ? \rangle)}, \\
 &\quad \cancel{(4, \langle X, Y, ?, ?, 0.0 \rangle)}, \cancel{(5, \langle X, Y, ?, 0.0, 0.0 \rangle)}, \cancel{(6, \langle X, Y, ?, 0.0, 0.0 \rangle)}, \\
 &\quad \cancel{(11, \langle X, Y, ?, Y, 0.0 \rangle)}, (12, \langle X, Y, ?, Y, 0.0 \rangle) \} \\
 &= \{12, \langle ? \rangle\}
 \end{aligned}$$

定义5 一个程序在一个切片标准 $C = (i, V)$ 上的一个切片 S (即 S 是程序 P 使用切片标准 C 之后获得的一个程序子集) 是任何满足下面两个属性的可执行程序。

属性1 S 可以从 P 中删除零条或多条语句获得；

属性2 对于轨迹 T ，无论何时程序 P 在输入 I 上阻塞，那么其切片 S 在轨迹 T' 上也会对输入 I 阻塞，并且 $Proj(C, T) = Proj(C', T')$ ，其中 $C' = (i', V)$ ，并且如果语句 i 在切片中，那么 $i' = i$ ，或者 i' 是 i 最近的后继语句。

我们知道，对于一个中等复杂的程序来说，要进行比较完整的测试其成本是非常高的。那么使用切片技术，可以把一个规模较大且较复杂的程序转换成多个使用一定切片标准而获得的切片程序，这些切片相对原来的程序来说，要简单且易于测试。

4.5 计算流视角

一个程序可以被看成一个有限“计算集”的表示。一个“计算”是当程序执行特定输入时产生的数据状态的轨迹。通过一次执行来促使一个程序的计算流 (computation flow) 被完整分析，可以服务于估计在代码中遗留缺陷的数量、测试数据捕捉缺陷的能力和程序隐藏缺陷的可能性。

图 4-7 给出的是一个计算二次方程的根的计算流图。

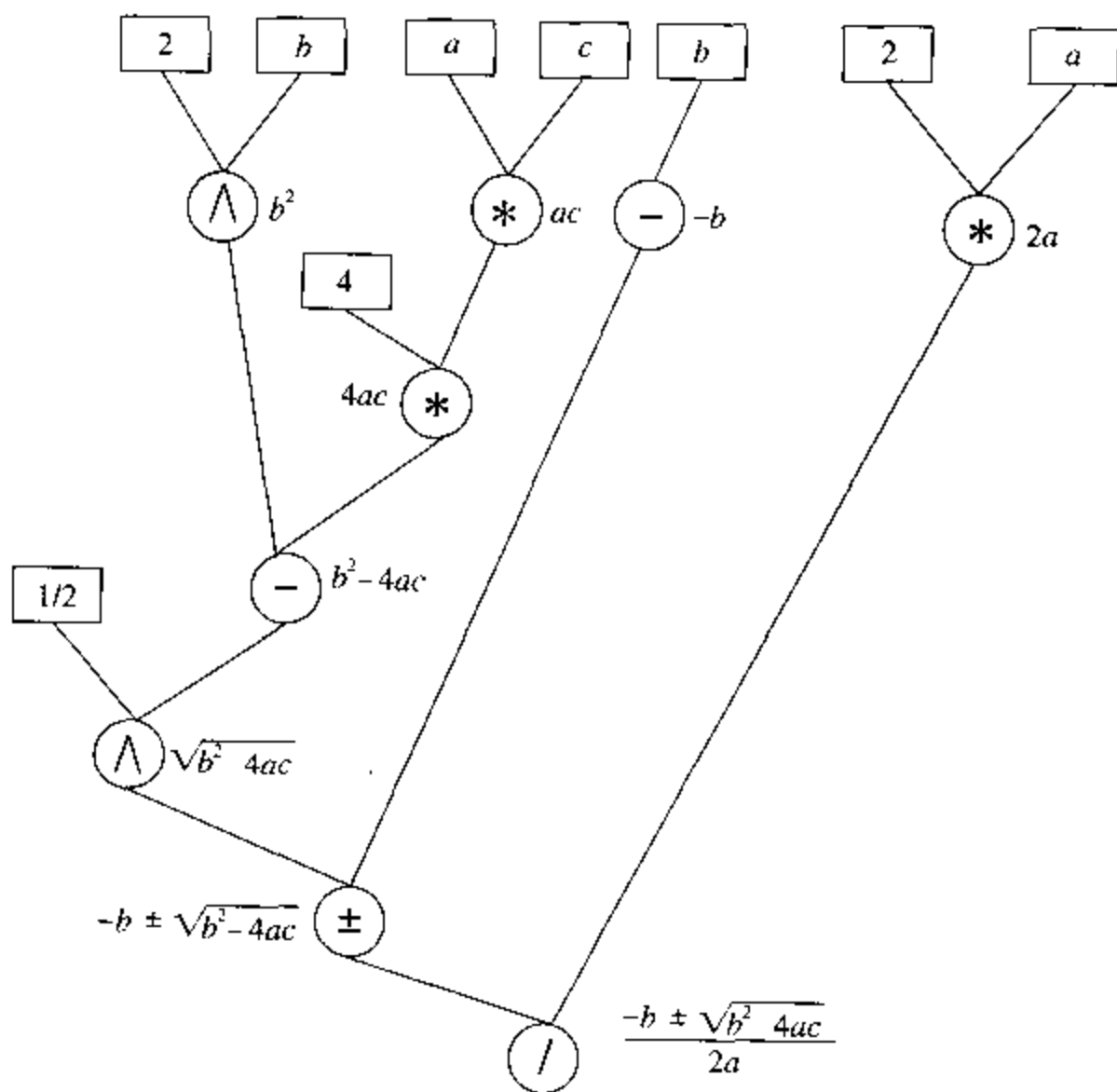


图 4-7 二次方程求根的计算流图

4.5.1 故障插入

故障插入(Fault Seeding)是一个统计的方法,用于评价遗留在一个程序中的故障数量和种类^[58]。首先,故障被插入到一个程序中,然后,程序被测试,并且发现故障的数量可用来估计还没有被发现故障的数量。计算公式如下:

原本错误总数 = (插入的错误总数/发现的插入错误数) × 发现的原本错误数

残留错误数 = 原本错误总数 - 发现的原本错误数

故障插入技术在软件可靠性方面应用比较广泛,尤其在硬件的测试当中。例如一些汽车公司不惜花大价钱进行汽车碰撞试验,为的就是发现汽车在碰撞过程中的潜在隐患,通过消除、改进这些隐患从而达到更高的性能。

借鉴硬件可靠性测试方法,在软件测试中也引入故障插入技术。其主要目的是为了评价系统的哪些模块、哪些代码是危险模块、危险代码,容易出问题,从而评价系统的容错能力。在该技术中使用了故障加速技术,通过有意插入的故障来调用系统的故障容错能力,从而在一个可控制的环境和期望的时间段内获得完整的测试。它和现有的测试方法相比,最大不同在于测试开始时的系统状态不同。现有的测试都是从系统的正确状态开始,测试系统如何转入故障状态,而故障注入测试则是从系统的故障状态开始,测试系统在发生故障后的运行规律。这里要重点指出的是,故障插入不关注为什么出现这样的故障,它关注的是出现了故障后系统如何处理。

故障插入也可用于验证测试用例的有效性。其原理就是为了检查设计的测试用例是否能发现某一类型的故障,人为地在被测系统中引入该类型的故障,如果在测试过程中能发现这个故障,则应该也可以测试出系统原来就存在的该类故障。

故障插入技术的一个难点是,插入的故障在程序中是否能够代表还没有发现的故障。实际上,由于软件的复杂性,故障的暴露往往和当时的运行环境、执行路径有很大关系。能测试出故障插入的故障并不一定总是能测试出被测系统原本存在的该类故障。所以,在上述的最后一步推论不一定总是成立。但是,就验证测试用例的有效性角度来看,故障插入确实可以作为一种手段。

关于在测试中如何具体使用该技术的内容请参考本书5.2.3节。

4.5.2 变体分析

变体分析(变异分析)(Mutation Analysis)使用故障插入来分析测试数据的属性^{[59][60][61][62]}。带有插入错误的程序称为一个变体或变异。执行变体以确定它们的行为是否不同于原来程序的行为。行为不同的变体假设已经被测试杀死。变体分析的一个产物是,关于测试数据能够在多大程度上杀死变体的度量。变体通过使用一个变体操作(Mutation Operator)产生。这种操作变更程序中的一个简单表达式成为另一个表达式。表达式的选择来自于表达式的一个有限类。例如,一个常量可能被增加1或减少1,或者用0来代替,产生三个变体中的一个。在程序任何可以使用这个操作的点上应用这个操作将产生一个有限的,很大的变体集合。

考虑下面的 C 语言程序：

```
main () /* compute sine function */
{
    int i;
    float e, sum, term, x;

    scanf ("%f %f", x, e);
    printf ("x=%10.6f e=%10.6f \n", x, e);
    term = x;
    for (i=3; i<=100 && term > e; i=i+2)
    {
        term=term * x * x / (i * (i-1));
        if (i % 2 == 0) sum = sum + term;
        else sum = sum - term;
    }
    printf ("sin(x)=%8.6f \n", sum);
}
```

下面是它的几个变体：

1. 把变量 x 设成常量 0，产生一个变体

```
main () /* compute sine function */
{
    int i;
    float e, sum, term, x;

    scanf ("%f %f", x, e);
    printf ("x=%10.6f e=%10.6f \n", x, e);
    term = 0;
    for (i=3; i<=100 && term > e; i=i+2)
    {
        term=term * x * x / (i * (i-1));
        if (i % 2 == 0) sum = sum + term;
        else sum = sum - term;
    }
    printf ("sin(x)=%8.6f \n", sum);
}
```

2. 把 $i \leq 100$ 写成 $i \geq 100$ ，产生一个变体

```
main () /* compute sine function */
{
    int i;
    float e, sum, term, x;

    scanf ("%f %f", x, e);
    printf ("x=%10.6f e=%10.6f \n", x, e);
    term = x;
    for (i=3; i<=100 && term > e; i=i+2)
    {
        term=term * x * x / (i * (i-1));
```

```

        if (i % 2 == 0) sum = sum + term;
        else sum = sum - term;
    }
    printf ("sin(x) = % 8.6f \n", sum);
}

```

3. 把常量0换成1, 产生一个变体

```

main () /* compute sine function */
{
    int i;
    float e, sum, term, x;

    scanf ("% f % f", x, e);
    printf ("x = % 10.6f e = % 10.6f \n", x, e);
    term = x;
    for (i = 3; i <= 100 && term > e; i = i + 2)
    {
        term = term * x * x / (i * (i - 1));
        if (i % 2 == 0) sum = sum + term;
        else sum = sum - term;
    }
    printf ("sin(x) = % 8.6f \n", sum);
}

```

上面涉及到了变体的操作。那么, 变体操作实际上是怎样构造的呢? 很明显, 变体操作与程序设计语言密切相关。R. A. DeMillo 等人给出了 Fortran 程序的变体操作^[61], 具体如下。

- 常量变换: 把一个常量, 假设是 C , 替换成 $C + 1$, 或 $C - 1$; 例如, 语句 $A = 0$ 变成了 $A = 1$ 或者 $A = -1$;
- 标量替换: 把标量变量替换成另一个, 例如, 语句 $A = B - 1$ 替换成 $A = D - 1$;
- 标量替换常量: 把一个常量用一个标量变量替换, 例如, 语句 $A = 1$ 替换成 $A = B$;
- 常量替换标量: 使用一个常量替换一个标量变量, 例如, 语句 $A = B$ 替换成 $A = 1$;
- 用源常量进行替换: 把程序中的一个常量替换成同一个程序中的另一个常量。例如, 语句 $A = 1$ 替换成 $A = 11$, 其中 11 出现在程序其他语句中;
- 用数组引用替换常量: 把一个常量替换成一个数组元素, 例如, 语句 $A = 2$ 替换成 $A = B(2)$;
- 用数组引用替换标量: 把一个标量变量替换成一个数组元素, 例如, 语句 $A = B + 1$ 替换成 $A = X(1) + 1$;
- 相似数组名字替换: 把一个带有下标的变量替换成有相同规模和维度的另一个数组的元素, 例如, 语句 $A = B(2, 4)$ 替换成 $A = D(2, 4)$;
- 用常量替换数组引用: 把一个数组元素替换成一个常量, 例如, 语句 $A = X(1)$ 替换成 $A = 5$;

- 用标量替换数组引用：把一个数组元素替换成一个标量变量，例如，语句 $A = B(1) - 1$ 替换成 $A = X - 1$ ；
- 用数组引用替换数组引用：把一个数组元素替换成另一个数组元素，例如，语句 $A = B(1) + 1$ 替换成 $A = D(4) + 1$ ；
- 求负操作插入：插入求负操作，诸如在任何数据引用前加上负号，例如，语句 $A = X$ 替换成 $A = -X$ ；
- 数学操作符替换：把数学操作符号（即 $+$ ， $-$ ， $*$ ， $/$ ， $**$ ）替换成另一个数学操作符号，例如，语句 $A = B + C$ 替换成 $A = B - C$ ；
- 相关操作符替换：把一个相关操作符（即 $<>$ ， $<=$ ， $<$ ， $>=$ ， $>$ ）替换成另一个，例如，表达式 $X = Y$ 替换成 $X <> Y$ ；
- 逻辑连接符替换：把一个逻辑连接符（即 $.AND.$ ， $.OR.$ ， $.XOR.$ ）替换成另一个，例如，表达式 $A .AND. B$ 替换成 $A .OR. B$ ；
- 删除求负操作：删除任何求负操作，例如，语句 $A = -B/C$ 替换成 $A = B/C$ ；
- 语句分析：把一个语句替换成一个会引起程序执行立即终止的陷阱语，例如，语句 $GOTO 10$ 替换成 $CALL TRAP$ ；
- 语句删除：从程序中删除一个语句；
- 返回语句：在子程序中使用一个 $RETURN$ 语句替换语句；
- $GOTO$ 语句替换：替换 $GOTO$ 语句的标号为另一个，例如，语句 $GOTO 20$ 替换成 $GOTO 30$ ；
- DO 语句 END 替换：替换一个 DO 语句的结束标号为其他标号，例如，语句 $DO 5 I=2, 10$ 替换成 $DO 40 I=2, 10$ ；
- 数据语句替换：更改使用一个数据语句分配的变量的值，例如，语句 $DATA Y / 22 /$ 替换成 $DATA Y / 33 /$ 。

变体分析在程序测试中的应用是相当广泛的，并且是一个相当重要的技术，可以用于验证测试的充分性，有关这方面的具体内容请参考本书 5.2.3 节。

4.5.3 敏感性分析

能够导致程序失败的故障有 3 个充分必要条件：执行 (Execution)、影响 (Infection) 和传递 (Propagation)^{[63][64]}。更清晰的解释就是，缺陷所在的位置必须要被执行到，缺陷被执行后必须要使程序的数据状态产生错误变化，并且这个错误的数据状态在程序后面的计算中不会被纠正，最终使程序产生一个失败。在附录参考书目的^[65]和^[63]中讨论了对于发生影响和传递的条件。

敏感性分析 (Sensitivity Analysis) 研究了故障需要的 3 个条件，尤其关注的是错误的影响和传递方面^{[35][64]}。影响分析 (Infection Analysis) 使用变体分析来确定一个潜在的故障语句被执行后，一个数据状态被影响的可能性。传递分析 (Propagation Analysis) 则变化数据状态来确定一个影响了的数据状态会产生一个程序失败的可能性。Voas 给出了一个方法用于估计执行、影响和传递会因为错误和特定类型数据状态而发生的可能性^[64]。Morell 使用符号执行来分析潜在的错误流^{[63][66]}。符号故障被引入到程序中，并且程序被符号化

执行。符号输出捕获故障对程序计算的影响。

4.6 功能视角

程序可以被看成是一个功能集, 功能被认为是对一个有序对 (x, y) 集合的指示。其中 y 是由程序在输入 x 处阻塞而产生的输出。用输入 x 执行程序并观察它的输出(如果有的话)是一种分析技术, 该技术提供了程序在给定输入上的直接证据。

符号分析

符号分析(Symbolic Analysis)探索使用一种更一般的方法来描述被程序计算的功能。一个符号执行系统(Symbolic Execution System)接受3个输入: 被解释的程序、用于程序的符号输入和沿循的路径。它产生两个输出: 描述已选择路径计算的符号输出和那个路径的路径条件^[67]。路径的规格可以是交互的^[68], 也可以是预先选择的^{[69][70]}。符号输出可以用于证明程序相对于它的规格的正确性, 并且, 路径条件可以用于产生测试数据以执行期望的路径。然而, 结构化数据类型会有所不同, 因为它有时不可能推断出什么组件被修改了。

路径的符号执行可以通过符号化地执行路径中出现的赋值语句来实现。这样, 赋值语句通过符号化的评价来执行赋值语句右边的表达式。最终的符号值变成了赋值语句左边变量的新的符号值。一个逻辑的或数学的表达式通过把其中的变量替换成符号值来执行。

出现在条件分支语句中的分支条件和谓词可以被符号化地执行以形成符号谓词。在符号路径执行期间, 用于一条路径的谓词符号系统可以通过同时符号化执行赋值语句和分支谓词来构造。谓词符号系统包含通过分支谓词执行而产生的符号谓词序列。所有的符号化执行系统都必须包含一种便利的技术(或工具)用于选择符号化执行的路径, 用于符号化地执行路径, 并且用于产生需要的符号化输出。

有3种类型的路径选择技术可以使用: 交互的、静态的和自动化的。在交互方法中, 用户事先指定要执行的路径, 然后符号化执行系统被自动构造。在符号化执行期间, 当每次必须决定执行那个分支时, 控制就被返回给用户。在自动化方法中, 符号化执行系统试图使用一个一致的谓词系统来执行所有的程序路径。

符号评价(Symbolic Evaluation)在检测计算错误方面是最有效的, 同时也能检测逻辑错误和数据处理错误。这个技术在帮助测试数据的开发方面也是很有用的。一个主要的应用是增加一个程序的信心级别。正确的符号输出表达式使测试人员相信代码执行了期望的计算。

符号评价适合于能使用简洁和形式化方法表示的语言。许多符号评价系统都是用于代数编程语言的, 例如Fortran。符号评价对小段代码特别有用。当代码规模、路径长度和程序变量增加时, 符号评价就变得成本更高了。同时符号评价要求测试人员有更高的技术。

4.7 本章小结

本章讨论了许多与分析相关的技术。根据 SEI 的分类, 这些技术被划分成 6 种不同的视角: 文本视角、句法视角、控制流视角、数据流视角、计算流视角和功能视角。每种视角所包含的信息呈递增趋势。文本视角把程序看成是字符和记号的序列。句法视角把一个程序看成是一个由程序语言语法决定的句法元素的一个分层结构。程序插装是句法视角的一个典型代表, 且在实际软件测试中有着重要的应用, 现在市场上很多用于覆盖率分析的工具基本都是基于该技术的。控制流关系叙述了程序元素和它们执行的次序之间的联系, 控制流视角的控制流视图是一个应用非常广泛的技术, 也是作为测试人员必须掌握的一种技术, 一个程序可以表示成一个数据流图, 使用变量定义、引用和未定义信息做标记。使用数据流分析技术可以发现变量定义和使用方面的异常。计算流视角把程序看成是一个有限计算集并表示, 一个计算是当程序执行一个特定输入时产生的数据状态的轨迹。计算流视角的典型应用包括故障插入、变体分析和敏感性分析。尤其是故障插入和变体分析在软件测试中的应用越来越广泛, 在测试充分性方面也得到了推崇。从功能视角看, 程序可以被看成是一个功能集, 功能被认为是对一个有序对 (x, y) 集合的指示, 其中 y 是由程序在输入 x 处阻塞面产生的输出。使用输入 x 执行程序并观察它的输出 (如果有的话) 是一种分析技术, 该技术提供了程序在给定输入上的直接证据。该视角的一个应用是符号化分析。

求规格说明书、概要设计文档和详细设计文档^[42]。其目标是为了证明需要的每个软件都已存在。软件特性包括输入域、输出域、应当受到相同处理的输入分类和处理功能本身。一个未实现的需求在代码中反应为丢失的路径或丢失的代码。面向规格的测试假定了一个软件功能视图,有时也叫功能测试或黑盒测试^[43]。

5.1.1 测试独立于规格技术

规格细化了用于一给定软件单元的假设。它必须描述接口。通过这些接口访问给定的单元。同时,这种给定的访问也需要被描述。一个单元的接口包括它的输入、输出和它们相关的值域空间。一个模块的行为总是包括计算功能(语义的)和运行时的特性。例如,它的空间和时间复杂度。面向规格的测试从规格的这些特性导出测试数据。

1. 基于接口的测试

基于模块接口的测试根据模块和它们相互关系的特性选择测试数据。

- 输入域测试

在极端测试(Extremal Testing)中,测试数据被选择用来覆盖输入域的极端情况。与此类似,中间范围测试(Midrange Testing)则选择域内部的数据进行测试。其动机是诱人的——它希望整个输入域的结果可以从行为中得到。这些行为由它的代表成员引出^[2]。对于结构化的输入域,每个成员的极端点的组合被选择。这个过程可以产生大量的数据,尽管输入域之间内部联系的考虑在一定程度上可以减轻这个问题。

- 等价类分析

规格经常把所有可能的输入集合分成一个-一个类。这些类中间的数据获得相同的对待。这个划分被称为等价类^[2]。等价类的结果是确认了功能的一个有限集合以及它们相关的输入和输出域。例如,规格:

$$\{(x,y) \mid x \geq 0 \supset y = x \ \& \ x < 0 \supset y = -x\}$$

把输入划分成两个集合,分别是相等的和否定的函数。输入约束和错误约束也可以从这个分类中产生。一旦这些分类被划分出来,极端测试和中间范围测试就可应用到结果输入域上。

等价类划分可有两种不同的情况:有效等价类(Valid Equivalence Class)和无效等价类(Invalid Equivalence Class)。有效等价类是指对程序规格说明是由合理的,有意义的输入数据构成的集合,利用有效等价类可检验程序是否实现了规格说明中所规定的功能和性能。无效等价类是与有效等价类定义恰巧相反的类。

实际划分等价类时需要充分分析代码的逻辑结构,分析各控制点与输入之间的关系,细化等价类的划分。同时,根据输入各参数和全局变量,划分出各等价类组合。测试用例可以从各组合中获取。当软件变得复杂时,等价类的区分以及相互之间的作用关系也相应复杂,不能方便地应用这种方法。下面给出一些等价类划分原则:

(1) 在输入条件规定了取值范围或值个数的情况下,可以确立一个有效等价类和两个无效等价类;

(2) 在输入条件规定了输入值集合或者规定了“必须如何”的情况下,可确立一个有

效等价类和一个无效等价类;

(3) 在输入条件是一个布尔量的情况下,可确定一个有效等价类和一个无效等价类;

(4) 在规定了输入数据的一组值(假定 n 个),并且程序要对每一个输入值分别处理的情况下,可确立 n 个有效等价类和一个无效等价类;

(5) 在规定了输入数据必须遵守规则的情况下,可确立一个有效等价类(符合规则)和若干个无效等价类(从不同角度违反规则);

(6) 在确知已划分的等价类中各元素在程序处理中的方式不同的情况下,则应再将该等价类进一步地划分为更小的等价类。

• 边界值测试

边界值测试(Boundary-Value Testing)是从输入域测试中衍生出来的。Mayer 定义了边界的条件,它的值落在等价类的边界上、边界外和边界内^[2]。边界值测试是对等价类测试的一个补充,但不同于等价类测试。长期的测试工作经验表明,大量的错误发生在输入或输出范围的边界上,而不是发生在输入输出范围的内部。因此,针对各种边界情况设计测试用例,可以查出更多的错误。使用边界值分析方法设计测试用例,首先应确定边界情况。通常输入和输出等价类的边界,就是应着重测试的边界情况。应当选取正好等于,刚刚大于或刚刚小于边界的值作为测试数据,而不是选取等价类中的典型值或任意值作为测试数据。

跟等价类划分测试一样,边界值测试在软件比较复杂时也会变得不实用。边界值分析对于非向量类型的值(如枚举类型的值)也没有意义。为了有效使用边界值分析法,下面给出一些边界值选择测试用例的原则:

(1) 如果输入条件规定了值的范围,则应选取正好达到这个范围的边界值,以及刚刚超越这个范围的边界值作为测试输入数据;

(2) 如果输入条件规定了值的个数,则用最大个数、最小个数、比最小个数少一、比最大个数多一的数作为测试数据;

(3) 根据规格说明的每个输出条件,使用前面的第一条原则;

(4) 根据规格说明的每个输出条件,应用前面的第二条原则;

(5) 如果程序的规格说明给出的输入域或输出域是有序集合,则应选取集合的第一个元素和最后一个元素作为测试用例;

(6) 如果程序中使用了一个内部数据结构,则应当选择这个内部数据结构的边界上的值作为测试用例;

(7) 分析规格说明,找出其他可能的边界条件。

• 语法检查

每个健壮的程序都必须分析它的输入并处理格式不正确的数据。验证这个特性称为语法检查(Syntax Checking)^[26]。实现的手段是使用一个广泛的数据来执行这个程序。通过使用BNF语法来描述数据,输入语言的实例可以使用自动化算法来产生。参考文档^{[44][45]}描述了一个使用有限控制系统来产生数据的例子。

2. 基于计算功能的测试

等价类导致功能与相关输入域和输出域的一个有限集确定。可以根据对这些功能的已

知特性开发测试数据。例如,考虑一个有固定点的计算功能,即它的某些输入值根据功能被映射到它们自身身上。在这些固定点上,测试这个计算是可能的,即使在没有完整规格情况下也可以^[47]。为了保证输出域的足够覆盖,对规格的了解是很关键的。

- 特殊值测试

根据计算功能特性的基础来选择测试数据被称为特殊值测试(Special Value Testing)^[10]。这个过程尤其适合于数学计算,计算功能的属性有助于选择能够表示被计算方案正确性的点。例如,sin 函数的周期建议使用不同于 2π 的任何倍数的测试数据。这个特性对于数学计算来说不是惟一不变的。例如,大部分好的打印系统当被应用到自身输出时,应当不做修改地再次产生它。一些字处理器也是这样的。

- 输出域覆盖

对于通过等价类确定的每个功能,都有一个相关的输出域。通过选择那些会导致每个输出域极端被达到的点来执行输出域覆盖(Output Domain Coverage)^[10]。这保证单元已经被检查了最大和最小的输出条件并且如果可能的话,所有错误信息的分类都已经被产生过了。一般来说,构造这种测试数据需要对计算功能很了解,因此是应用领域内的专家。同时,类似的输入域测试,输出域覆盖也可以结合边界值分析的方法。

5.1.2 测试依赖于规格技术

使用规格技术可以帮助测试。一个可执行的规格可以被作为一个“圣典”(Oracle),并且在某些情况下作为测试数据的一个生成器。一个规格的结构化属性可以指导测试过程。如果落在某个限定类的范围内,那些类的属性可以指导测试数据的选择。

1. 规范导出

规范导出(Specification Exporting)测试根据相关规范描述来设计测试用例。每一个测试用例用来测试一个或多个规范陈述语句。一个比较实际的方法是根据陈述规范所用语句的顺序来相应地为被测对象设计测试用例。

例:一个计算平方根函数的规范可以表达如下。

输入:实数

输出:实数

规范:当输入一个0或比0大的数时,返回其正的平方根;当输入一个小于0的数时,显示错误信息“平方根非法——输入值小于0”,并返回0;库函数 Print_Line 可以用来输出错误信息。

在这个规范中有3个陈述,可以用两个测试用例来对应:Print_Line 用来传送错误信息。

测试用例1:输入4,输出2

对应规范中的第一句陈述:“当输入一个0或比0大的数时,返回其正的平方根”。

测试用例2:输入-1,输出“平方根非法——输入值小于0”

对应规范中的第二、第三句陈述:当输入一个小于0的数时,显示错误信息“平方根

非法——输入值小于0”，并返回0；库函数 Print_Line 可以用来输出错误信息”。

规范导出测试用例和规范陈述之间做到了很好的对应，加强了规范的可读性和可维护性。但它是一种正向测试用例的设计技术，所以还需要逆向测试技术来对测试用例进行补充，以达到更充分的测试。规范导出测试的变化形式可以应用到保密分析、安全分析、软件故障分析或其他对系统规范做出补充的文件上去。

2. 代数方法

在代数规格 (Algebraic Specification) 中，一个数据抽象的属性通过公理或者重写规则表达出来。在测试系统 DAISTS (Data- Abstraction, Implementation, Specification and Testing System) 中，一个代数规格和它的实现之间的一致性被通过测试检查^[47]。每条公理被编译成一个过程，然后把它和一组测试点相联系。一个驱动程序提供这些点中的每个点到它所代表的公理过程中，过程依次指出公理是否被满足，实现和规格的结构化覆盖率同时被计算。Jalote 和 Pankaj 讨论了一种方法来产生测试数据验证一个代数规格的完整性^[96]。代数方法还可以应用到面向对象程序测试中。

3. 公理方法

尽管使用谓词积分 (Predicate Calculus) 作为规格语言已经相当广泛，但是从这个规格中引出数据的文献还很少。Gourlay 和 John S 在关于谓词积分和路径测试之间关系方面做了一些研究^[84]。

4. 状态机

许多程序可以表示为状态机 (State Machine)。这样，就提供了另一种选择测试数据的手段^[26]。既然两个有限自动机的相同问题是可以判定的，那么测试可以用于判定一个使用节点边界值模拟的有限状态机程序是否等同于一个指定的程序。这个结果可以用于测试那些可以通过有限状态机指定的程序的特性，例如，一个事务处理系统的控制流。

状态机分为有限状态机 (Finite State Machine, FSM) 和无限状态机 (Infinite State Machine, ISM)。有限状态机的实际使用比较广泛。一个有限状态机包含有限多个状态数量，并且在接受到输入后产生输出^{[98][99]}。有限状态机在许多领域被广泛应用到模型系统中，包括连续电路，通信协议等^[100]。由于有限状态机的实际重要性和理论的趣味性，因此关于有限状态机的测试问题已经在许多领域被研究很多次了^[97]。

有两种类型状态机模型：Mealy 模型和 Moor 模型。在 Mealy 模型中，一个转换可能有一个输出动作，且任何一个输出动作都可被用在不止一个转换上^[117]，输出和状态没有联系，即状态是被动的。在 Moor 模型中，转换没有输出动作^[118]，一个输出动作和每一个状态联系起来，即状态是主动的。图 5-1 展示了 Mealy 模型与 Moor 模型的比较。由于 Moor 模型比较复杂，因此 Mealy 模型用得比较广泛。

有限状态机的定义可以表示成如下公式：

$$M = (I, O, S, \delta, \lambda)$$

其中： I 表示有限且非空的输入信号集合；

O 表示有限且非空的输出信号集合；

S 表示有限且非空的状态集合；

δ 表示 $S \times I \rightarrow S$ 状态迁移函数；

λ 表示 $S \times I \rightarrow O$ 输出函数。

当状态机的当前状态 s 在 S 集合中，且从 I 中接受输入 a ，那么状态机的状态将切换到 $\delta(s, a)$ 并且产生输出 $\lambda(s, a)$ 。

状态机测试主要关注于测试状态转移的正确性。对于一个有限状态机，通过测试验证其在给定条件内是否能够产生需要的状态变化，有没有不可达的状态和非法的状态，会不会产生非法的状态转移等。

5. 判定表

判定表(Decision Tables)是表示一个等价类的简明方法。判定表的行指定所有输入可能满足的条件，判定表的列指定可能发生的不同行为的集合。这个表的入口表示如果条件被满足，行为是否应当被执行。典型的入口为“是”，“不是”或者“忽略”。表的每一行建议使用重要的测试数据。

一个有限入口的判定表(Limited-Entry Decision Table)格式见图 5-2。

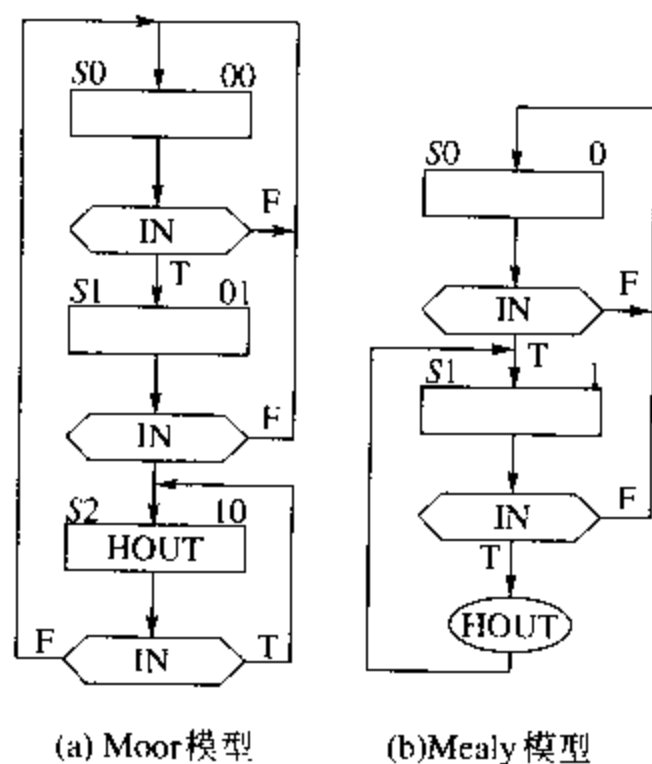


图 5-1 Moore 模型和 Mealy 模型

条件桩	条件入口
动作桩	动作入口

图 5-2 有限入口判定表格式

其中：

- 条件桩表示条件的列表，每个条件一行；
- 条件入口列出了在一列中条件值的组合；
- 条件约束表达了条件之间的交互；
- 动作桩包含了动作的列表，每个动作一行；
- 动作入口对每个被执行的动作用“X”标记；
- 应用约束表达了一个动作被执行情况下的条件。

我们来分析下面这段规格。

写一段程序来重新格式化文本，具体如下。

给定一个文本，以 ENDOFTEXT 字符结尾，并且包含的字使用 BLANK 或者 NEWLINE

字符隔开, 根据下面规则把该文本转成一行接一行的格式:

- (1) 行在文本有 BLANK 或 NEWLINE 处中断;
- (2) 行尽可能长;
- (3) 没有行可以有超过 MAXPOS 长度的字符。

这段规格对应的判定表如下表 5-1。

表 5-1 判定表例子

	1	2	3	4	5	6	7	8	9	10
条件 1: 输入字符是 BLANK 或者 NEWLINE	Y	Y	Y	Y	N	N	N	N	N	N
条件 2: 输入字符是 ENDOF-TEXT	(N)	(N)	(N)	(N)	Y	Y	Y	Y	N	N
条件 3: 当前字非空	Y	Y	Y	N	Y	Y	Y	N	(Y)	-
条件 4: 当前输出行对当前字仍旧有足够空间	Y	Y	N	-	Y	Y	N	-	(N)	-
条件 5: 当前输出行非空	Y	N	-	-	Y	N	-	-	-	-
条件 6: 当前字有最大长度	(N)	(N)	-	(N)	(N)	(N)	-	(N)	Y	N
动作 1: 写 BLANK 并且增加当前字长度	X				X					
动作 2: 写 NEWLINE 并且重新设置当前字长度为 0		X	X			X	X			
动作 3: 写当前字	X	X	X		X	X	X			
动作 4: 告警									X	
动作 5: 加入字符到当前字后面并且增加当前字长度										X
动作 6: 读一个字符并且重复这个表	X	X	X	X						X
动作 7: 退出这个表					X	X	X	X	X	

其中: “-”表示要么 Y 要么 N; (Y) 或者 (N) 表示隐含的。

6. 因果图

因果图(Cause-Effect Graphing)^[2]提供了一个把规格转化为判定表的系统化方法, 从图 5-3 中, 测试数据可以被产生。其中, 原因表示输入条件, 结果是执行的一系列计算。

在因果图中有 4 种基本符号, 分别表示了规格的 4 种因果关系, 见图 5-3。

在因果图中使用了基本的逻辑符号, 图 5-3 左边的圈表示输入原因, 右边的圈表示输出结果。在实际问题中, 输入原因之间还可能存在某些依赖关系, 我们称它为“约束”(Constrain)。对于输入的约束有以下几种。

- E(Exclusive, 异或)约束 即在输入 a 和 b 中只能有一个为 1, 要么是 a , 要么是 b ;
- I(Inclusive, 包含)约束 即在输入的条件中, 必然有一个为 1;
- O(One and Only, 惟一)约束 即在输入条件中有且只有一个 1;

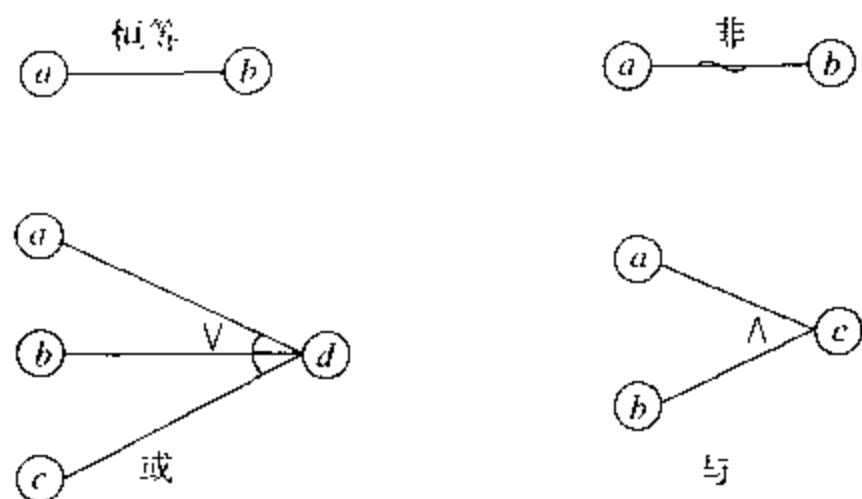


图 5-3 因果图基本符号

- R(Requires, 必备)约束 即如果 a 是 1, 则 b 必然是 1。

对于输出的约束只有一个:

- M(Masks, 掩码)约束 即如果 a 是 1, 则 b 是 0。

图 5-4 反应了这些约束的图形符号。

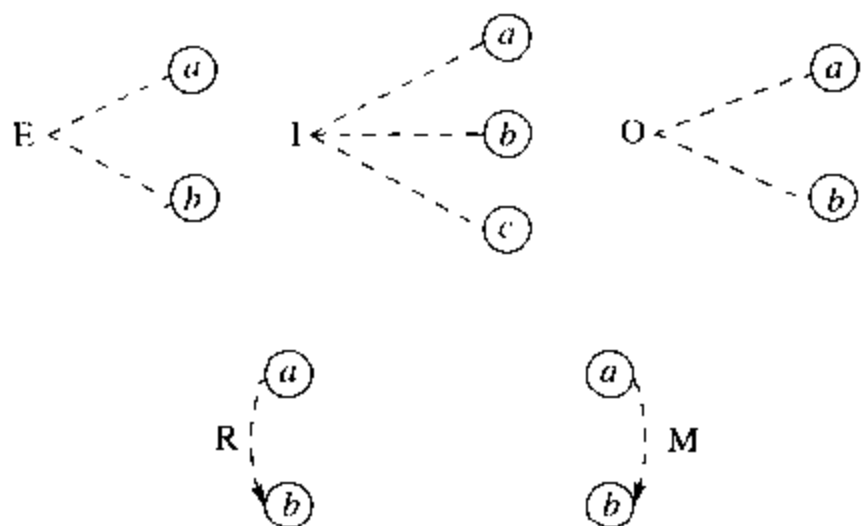


图 5-4 输入输出约束图形符号

根据因果图产生测试用例的步骤如下:

- S1, 把程序的规格划分解成可以工作的片段;
- S2, 确定规格中的原因和结果;
- S3, 分析规格以确定原因和结果之间的逻辑关系, 并且使用因果图表示出来;
- S4, 确定句法或环境的约束, 这些约束使得某些组合不可能产生;
- S5, 把因果图转化成有限入口判定表(参考 5.1.2);
- S6, 从判定表的每一列选取一个测试用例。

再看下面的例子^[2]:

“第一列字符必须 A 或 B, 第二列字符必须是个数字, 在此情况下文件被更新。但如果第一个字符不正确, 那么信息 X12 被产生; 如果第二个字符不是数字, 则信息 X13 被产生。”

在这个规范中, 原因有:

- 1 —— 第一个字符是“A”
- 2 —— 第一个字符是“B”

3 —— 第二个字符是一个数字

其结果是:

70 —— 进行更新

71 —— 产生信息 X12

72 —— 产生信息 X13

根据以上信息, 得到因果图, 如图 5-5 所示。

该因果图对应如图 5-6 所示的逻辑电路。

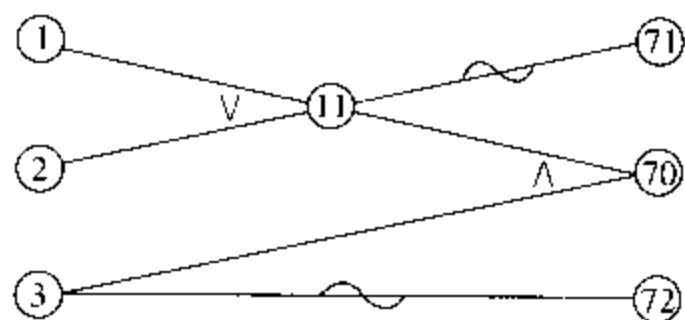


图 5-5 简单的因果图

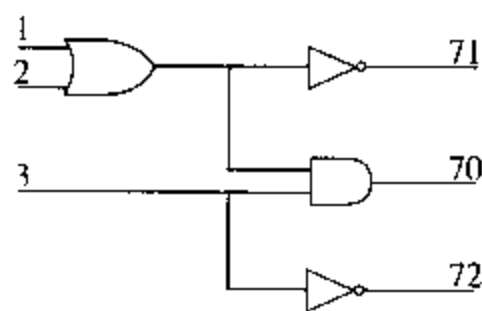


图 5-6 等价的逻辑电路图

考虑到原因 1 和原因 2 不可能同时成立, 这里可以使用 **E 约束**, 图 5-7 是带有约束的因果图。

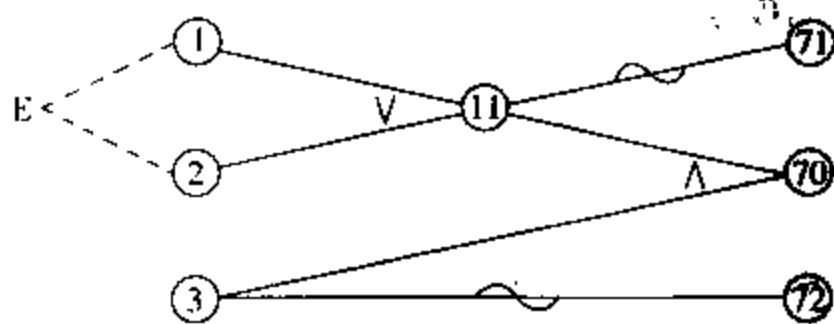


图 5-7 带有约束的因果图

使用前面的有限入口判定表知识, 可以得到如表 5-2 所示的判定表。

表 5-2 判定表

	1	2	3	4	5	6	7	8
条件 1: 第一个字符是“A”	1	1	1	1	0	0	0	0
条件 2: 第一个字符是“B”	1	1	0	0	1	1	0	0
条件 3: 第二个字符是一个数字	1	0	1	0	1	0	1	0
组合条件 11: (条件 1) V (条件 2)	1	1	1	1	1	1	0	0
动作 1: 进行更新	无	无	1	0	1	0	0	0
动作 2: 产生信息 X12	无	无	0	0	0	0	1	1
动作 3: 产生信息 X13	无	无	0	1	0	1	0	1
可能的测试用例	无	无	A5	Ax	B1	Bx	C3	Xx

7. 正交实验设计

无论使用因果图或者等价类划分方法, 有时很难从软件需求规格说明书中获得。而

且,即使对于一个中小规模的软件来说,要画出它的因果图也是相当困难且庞大的。根据这种因果图产生的测试用例其数目也是惊人的。为了有效地减少测试工作量,同时又尽可能不降低测试质量,可以使用正交实验设计方法。

所谓正交实验设计方法是指,从大量的实验点中挑选出适量的、有代表性的点,应用 Galois 理论导出正交表,合理进行实验安排的一种科学实验方法。利用这种方法,可使所有的因子和水平在实验中均匀分配和搭配,均匀地有规律地变化。在正交实验设计中,通常把判断实验结果优劣的标准叫做实验指标,把所有可能影响实验指标的条件称为因子,而影响实验的因子叫做因子水平。在进行实验优化设计时,为了完成明确的实验目的,必须有合理的实验指标,加上合理的标准来挑选实验因子以及相应的水平。

软件功能测试,作为实验的一种,完全可以利用正交实验设计方法来进行测试数据的选择,以提高测试效率。首先根据被测软件的规格说明书找出影响其功能实现的操作对象和外部因素,把它们作为因子,而把各个因子的取值当作状态,构造出二元因素分析表。然后,利用正交表进行各因子状态的组合,构造有效的测试输入数据集,并因此建立因果图。这样得出的测试用例集将大大减少。

关于该方法的详细技术可以参考郑人杰的《计算机软件测试技术》一书^[1]。关于正交的原理在各种测试方法中都被用到,例如 David M. Cohen 等人提出结合正交矩阵方法自动生成测试用例的 AETG 方法^{[101][102]}, McCabe 利用正交原理寻找最小正交路径组等^{[103][104]}。

8. 功能测试

功能测试的基本思想是假设 f 是程序 p 要计算的功能,并且, f^* 是 p 预期的功能,即描述在程序规格说明书中的功能。此外,假定存在一个功能集合 F_f ,它具有如下属性:

- f 在 F_f 中;
- f^* 在 F_f 中;
- 对于 f 或 f' (f' 在 F_f 中),存在一种选取测试的方法,在该方法选取的测试中,如果 f 和 f' 都能满足这些测试,那么就认为它们是等价的。

Howden 提出的功能测试方法是根据属性 3 来选择测试数据的^[105]。在该方法中,它假定有 3 种类型功能:表达式功能、条件功能和迭代功能。

在表达式功能中,可以使用操作来组合一组函数成一个新的函数。例如,如果 $f_1(X)$ 和 $f_2(X)$ 是两个函数,那么它们可以组合成一个表达式,例如 $(f_1(X) + 3f_2(X))$,来形成一个新的函数。

一个布尔函数 $b(X)$ 和两个其他函数 $f_1(X)$ 和 $f_2(X)$ 可以用来组合成一个条件功能,如: $\text{if } b(X) \text{ then } f_1(X) \text{ else } f_2(X)$ 。

一个迭代功能是一个循环结构格式,具体如下:

```
repeat while  $b(x) = \text{true}$ 
     $x \leftarrow f(x)$ 
endrepeat
```

其中 \leftarrow 表示分配。

Howden 给出了下面这些选择测试用例的规则^[105]:

- 对于表达式功能

(1) 选择表达式中每个单个的变量都有非零的值, 并且表达式结果也是一个非零值;

(2) 假定表达式中的变量是 x_1, x_2, \dots, x_n ; 那么选择 $n+1$ 个测试用例 $x_{i,1}, x_{i,2}, \dots, x_{i,n}$, 其中 i 在 $1 \sim n+1$ 之间, 它们的选择要使得下面这个矩阵成为非单数矩阵(non-singular):

$$\begin{array}{cccccc} x_{1,1} & x_{1,2} & \dots & x_{1,n} & 1 \\ x_{2,1} & x_{2,2} & \dots & x_{2,n} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{n+1,1} & x_{n+1,2} & \dots & x_{n+1,n} & 1 \end{array}$$

- 对于格式为 if b then s1 else s2 的条件功能

(1) 假定谓词 b 有 (exp rel 0) 格式, 其中 rel 是 $<, <=, >, >=, <>, =$; 并且 exp 是一个数学表达式。假定, 要么 exp 是某个添加的常量的一个偏移 (即, 应当是 $\text{exp} + k$, k 不等于 0), 要么 rel 是一个错误的关系操作符。那么这个故障可以使用测试用例检测出来, 并且这些测试用例引起 exp 呈现最大负值、零和最小正值。此外, 对于通过条件来选择两个可相互替换的功能, 必须给测试用例不同的值。

(2) 如果谓词 b 有格式 $\text{exp1 rel1 0 and exp2 rel2 0 and } \dots \text{ and expn reln 0}$, 或者 $\text{exp1 rel1 0 or exp2 rel2 0 or } \dots \text{ or expn reln 0}$ 。那么选择测试用例以便在组合表达式中的每个条件 expi reli 0 被使用满足第 1 条规则的用例同时测试, 对于联结谓词 (and), 所有条件都得保持真值, 对于非联结谓词 (or), 所有条件都得保持假值。

(3) 让 exp' 是表达式 exp 中由于一个潜在错误变量替换而导致的表达式。选择满足第 1 和第 2 条规则的测试用例, 此外选择用于潜在错误变量的值要么在所有 3 个测试上 $\text{exp}' < 0$ 或者 $\text{exp}' > 0$; 要么 exp' 在 3 个测试上的值 ($\text{exp}'1, \text{exp}'2, \text{exp}'3$) 是 exp 在 3 个测试上的值 ($\text{exp1}, \text{exp2}, \text{exp3}$) 的一个左移排列或者右移排列。

- 对于结构如下的迭代功能

```
repeat while /until b(x) = true
    S1
    S2
    ⋮
endrepeat
```

(1) 选择一组测试用例 T, 这样对于每个测试, 在循环中被分配值的变量中的一个或多个在每个循环迭代期间它们的值被改变;

(2) 假定 b 有格式 exp rel 0 , 选择一个测试用例集合 T, 它导致 exp 取最大负值、零和最大正值, 并且, 在循环中代码的执行会对每个 T 中的测试用例改变其至少一个输出变量的值。

5.2 面向实现的测试

在面向实现的测试当中, 通过从实现中获取的信息来指导测试数据的选择^[48]。目的

是为了保证软件的不同计算特性被充分地覆盖。它希望满足这些标准的测试数据能够高效地发现缺陷。程序的每次执行经过一条特定的路径。因此,面向实现的测试关注下面几个问题:希望获得什么样的计算特性?这个程序的哪些路径可以获得这些特性?什么数据可以执行这些路径?由给定数据执行的路径集合的可计算特性是什么?

面向实现的测试表述了这样一个现实,即只有程序的文本揭示了程序的详细判定。为了有效测试程序,一个程序员可能会选择实现一些特殊的用例,这些用例不会出现在规格书中。相应的代码在使用面向规格的测试时,只是偶尔被测试到,然而使用结构化覆盖率度量应当可以揭示这种情况需要的测试数据(关于覆盖率的概念请参考本书第3章内容)。

面向实现的测试策略可以根据两个正交轴进行分类:面向错误和程序视角,这个我们已经在第4章讨论过了。一个测试的面向错误策略强调发现错误的方面:执行、影响或者传递(参考4.5节内容)。一个测试的程序视角策略是程序的抽象源被用于决定期望的计算特性:控制流、数据流或者计算流。程序视角强调在策略背后的动机并且帮助你更好地评价策略所做出的承诺。

下面几节内容根据面向错误的策略进行组织。需要特定程序元素执行的技术被首先介绍,然后是那些试图加强影响和那些试图加强传递的策略。应当被注意的是影响和传递技术都需要执行,并且一些技术强调所有三个条件(执行、影响和传递)。在每一节中,技术根据程序的视角进行排序:控制流、数据流和计算流。

5.2.1 面向结构的测试

一个测试技术如果是寻找会引起程序的不同结构特性被执行的测试数据,那么我们称它是面向结构的。评价获得的覆盖率可以包括插装代码以保持对程序在实际测试中哪些代码被执行的跟踪。这种插装技术的廉价成本已经成为采用面向结构技术的主要动机^[49]。进一步的动机是为了降低用户在没有被测试到的代码部分发现缺陷的可能性。

在面向结构的测试中,有3个主要的组成:语句、分支和数据。下面我们来分别讨论。

1. 语句测试

语句测试需要每个语句在程序执行过程中被执行到。尽管很明显,100%的语句覆盖率不能保证程序的正确性,但是同样是很明显的,任何在测试中没有被执行过的代码肯定更不能保证正确。语句测试是基于对语句覆盖率的理解面进行的,根据未执行的代码来产生测试数据。关于语句覆盖的概念参考本书3.2.1节的内容。

2. 分支测试

100%的语句覆盖并不能保证程序流程图中的所有分支被执行到。例如,执行一个if...then语句,由于没有else,因此当被测条件为真时,语句都能覆盖到,但是隐含的else语句就没有被执行到。分支测试主要要保证每个分支被执行到(参考本书3.2.2节的内容)。分支覆盖可以通过在控制点上插入一个探针(一段检测代码)来检测^[49]。这个插装对于语句覆盖同样是足够的(参考本书4.2.1节内容)。

3. 数据覆盖测试

在一些程序中,流控制的一部分是由数据决定的,而不是根据代码决定的。基于知识的应用,一些人工智能应用和表格驱动代码都是这种现象的例子。数据覆盖寻找能够保证数据的不同组成被执行到,即它们在执行时通过解释器被引用到或修改到。并行语句测试能够保证每个数据位置能够被访问到。此外,在基于知识的领域内,数据项可以按不同的次序访问,因此覆盖所有访问次序是很重要的。这些访问次序不同于分支覆盖。

5.2.2 面向影响的测试

一个测试如果搜索建立适合于影响的条件以便在潜在故障的位置发生,我们称这种测试技术是面向影响的,本节主要是描述一些测试数据来加强存在故障影响的测试技术(即错误放大技术)。

1. 条件测试

在条件测试中,在每个条件中的每个子句被强制呈现每个可能的值和其他子句进行组合^[41]。这样条件测试包含了分支测试。用于条件测试的插装可以通过把条件组合拆分成简单条件和嵌套if语句的方法来完成。这就把条件覆盖的问题简化为更简单的分支覆盖问题,用于控制流视图的算法也可以被使用。

2. 表达式测试

表达式测试(Expression Testing)需要每个表达式在测试期间呈现许多不同的值。以这样一种方式进行,没有一个表达式可以被更简单的表达式替代^[59]。如果假定每个语句包含一个表达式并且条件表达式组成了所有程序表达式的一个适当的子集,那么这个测试形式正好可以使用前面提到的测试技术。表达式测试需要运行时的支持以便进行插装^[60]。

3. 域测试

一个程序的输入域可以根据哪些输入会引起每个路径被执行来进行划分。这些划分被称为路径域(Path Domain)。引起一个输入与错误路径相关的故障被称为域故障(Domain Faults)。其他故障被称为计算故障(Computation Faults)。域测试(Domain Testing)的目标是发现域故障,主要通过保证测试数据限制未检测故障的范围^[71]。通过选择接近路径域边界的输入来完成域测试。如果边界是不正确的,这些点增加了影响产生的机会。域测试假定巧合的正确性(Coincidental Correctness)是不会产生的,即,它假定如果输入沿着一条错误的路径,那么程序会失败。Clarke优化了这种方法的故障检测能力^[72]。

域测试的策略是基于对输入域空间的一个几何分析,并且使用这样一个事实——在输入空间边界附近的点是最容易出错的。那么输入域是怎么被定义的呢?

从抽象角度来看,一个程序的功能可以表示成一个函数 $f: X \rightarrow Y$ 。一般来说, f 的定义可以表示成子函数 f_1, f_2, \dots, f_m 的集合,其中, $f_i: X_i \rightarrow Y$ (即, f_i 是函数 f 限制在输入 X_i 上的功能集合), $X = X_1 + X_2 + \dots + X_m$,并且如果 i 不等于 j ,那么 f_i 就不等于 f_j 。我

们可以使用 $f(x)$ 表示 f 在 x 处的值, 其中 x 属于 X 。

我们可以使用 (P, S) 来表示程序, 其中 P 是程序会被执行的条件, S 是程序执行的序列语句。假定 D 是程序所有可能的输入集合。那么这个程序有效的输入域应当是 $X = \{x \mid x \in D \wedge P(x)\}$ (即使 $P(x)$ 条件为真的所有输入集合), 并且程序应当有 n 条路径, 即:

$$(P, S) = (P_1, S_1) + (P_2, S_2) + \cdots + (P_n, S_n)$$

因此对于每个 $1 \leq i \leq n$, S_i 是被设计用于实现功能 f_i 的一系列语句, 其中 $1 \leq j \leq m$ 。注意 m 不一定等于 n 。

对于输入域 X 来说, 它是多维的, 其维度是输入变量的个数。输入空间结构是输入空间的一个几何表示。在给定的输入以及加在相应输入变量上的最大化和最小化约束下, 就可以构造一个输入空间结构。该结构给出了一个输入变量域的图形表示。

测试点被产生用于每个被确认的边界段。这里边界段是输入空间被程序中谓词界定的一部分。这些测试点决定关系操作符和边界位置是否正确, 如图 5-8 所示。

在图 5-8 中, 我们假定 a 、 b 是在给定的边界上, 如果测试要成功, 那么 a 和 b 要被定义用于域 D_i 的功能计算, 而 c 要被用于域 D_j 的功能计算。可能会有这样一种情况, 即正确的边界会与 ac 和 bc 交叉, 但不在 c 点上。为了验证正确的边界和给定的边界是一

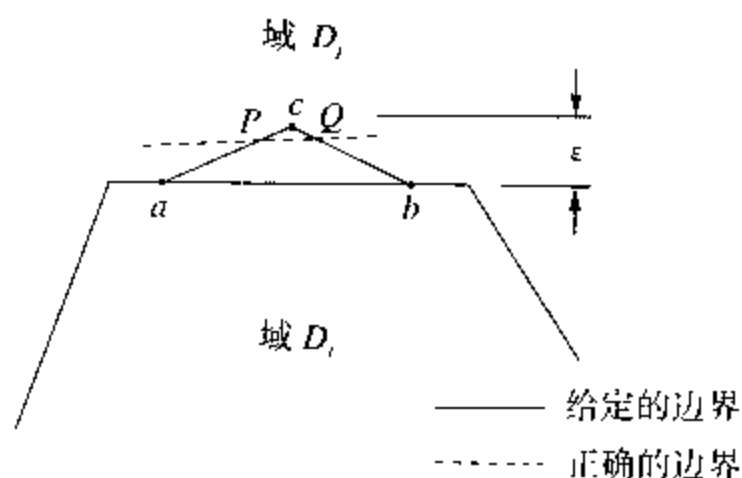


图 5-8 域测试点选择

致的, 可以选择一个点 c , 它到给定边界的距离是 ε , ε 是一个随意选取的很小的数字。如果 a 、 b 、 c 点能够满足给定域功能的计算, 那么给定的边界与正确的边界一致, 否则就可以检测出边界错误了。

对于域测试, 错误敏感的测试点可以通过下面的步骤产生:

- 选择程序中可能出错的谓词;
- 假定正确的谓词被确认;
- 把使用原始谓词的程序的结果和使用假定正确谓词的程序的结果进行比较;
- 导致结果不一致的测试点是对域错误敏感的测试点。

测试点的数量随着输入空间的维度和路径谓词数量线性上升, 同时, 为了进行域测试, 对程序提出了过多的限制。这是域测试的两个致命弱点。

域测试可以有效地检测条件语句中的操作数和操作符错误, 一般该技术应用于单元测试, 并且程序被使用代数语言或类 Pascal 语言编写; 控制结构应当精简。目前关于域测试的研究还在进行阶段, 这也是域测试还未被广泛使用的一个原因。

4. 扰动测试

扰动测试 (Perturbation Testing) 试图决定一个足够的路径集合来测试代码中的不同故障。故障作为一个向量空间被模型化, 并且特性理论描述了什么时候已经测试了足够的路径来发现计算错误和域错误。如果它们不能够减少错误空间的维度, 那么就不需要测试额

外的路径^{[73][74]}。

5. 故障敏感性测试

Foster 和 Kenneth A. 描述了一种选择测试数据的方法, 这些数据对故障非常敏感^[75]。Howden 使用了一种称为弱变体测试(Weak Mutation Testing, 参考 5.2.3 节内容)的方法对其进行形式化^[76]。用于确认错误敏感数据的规则被用来描述每个主要语言结构。在测试期间, 用于一个给定结构的规则的符合性意味着那个结构所有可供选择的形式已经被区分了。这对变体测试来说有一个明显的好处——消除所有没有产生一个错误的变体。一些规则甚至允许用于无穷多变体。

5.2.3 面向传递的测试

如果一个测试技术搜索保证潜在影响能够传递到失败的方法, 那么它被认为是面向传递的。这需要被选择的路径基于它们传递的特性进行测试。

1. 路径测试

在路径测试中, 数据被选择用于保证程序的所有路径已经被执行了^[76]。当然, 事实上, 这种覆盖率是不可能达到的(请参考本书 3.2.5 节内容)。第一, 任何有无限循环的程序有无限多条路径。第二, 存在不可达路径。第三, 一个不定的程序会在一个不定的输入上阻塞是不可决定的。

由于有上述困难, 一些简化的方法被建议使用。无限多路径可以根据循环的特点被划分成一个等价类的有限集合。边界和内部测试需要执行循环 0 次或 1 次(参考本书 3.5.7 节的内容), 并且, 如果可能, 执行最大可能的次数。线性代码和跳转标准(LCSAJ; Linear Sequence Code and Jump Criteria, 参考本书 3.5.9 节内容)表达了一个连续的更复杂的路径覆盖层次。很多文献都建议研究路径测试充分性的方法。

路径覆盖不能包含条件覆盖或者表达式覆盖, 因为一个表达式可能出现在多个路径上, 但是子表达式可能一直没有超过一个值(参考本书 3.2.5 节内容)。

由于要达到路径覆盖几乎是不可能的, 因此如何选择测试执行的路径对测试的有效性相当关键, 路径的选择可以结合数据流分析技术一起使用。关于数据流分析及测试方面的技术可以参考本书 4.4 节以及 5.2.3 节的内容。

• 线性独立路径

McCabe 根据循环复杂度来选择一个最大的线性独立路径(Linearly Independent Path)集合进行测试^[103]。McCabe 的循环复杂度是以图论为基础的。如果一个图的任何一个点都可以到其他任何一个点, 我们称其是强连接的(Strongly Connected)。在一个强连接图 $G = \langle E, N \rangle$, 其中 E 是 G 中边的集合, N 是 G 中节点的集合。那么该图中的线性独立路径最大集合是 $V(G)$, 可以表示成:

$$V(G) = E - N + 1$$

$V(G)$ 也叫 McCabe 循环复杂度。

这里我们来分析一个带有一个入口和一个出口的程序。它有这么一个属性, 即每个节

点都可以从入口处被到达,并且每个节点都可以到达出口点。但一般来说,它不是强连接的,因为无法从流程图上处于后面的点到达前面的点,例如从出口点到入口点。因此,为了达到强连接,McCabe 提出在控制流图的出口点和入口点处添加一条有向边,如图 5-9 所示,这样对于程序的控制流图来说,其 $V(G) = E - N + 2$ 。

关于如何从图中选择线性独立路径集合的详细方法请参考笔者即将出版的书籍《软件测试技术研究》。

- All-du-path 测试

在介绍下面这些路径选择标准之前,先介绍一些概念。

对于一个变量,假设为 x ,如果一条路径起始于 x 被定义的点,并且不包含任何使得变量 x 被取消定义或者重定义的语句,那么我们称这条路径是定义清晰的 (Definition Clear)。

如果路径中的每个节点只出现一次,那么我们称该路径是无循环的 (Loop Free)。

如果一条路径中最多只有一个节点出现过两次,那么该路径是简单路径 (Simple Path)。

一个变量,假设 x 的 du path 是一条简单路径,它对 x 是定义清晰的。

All-du-path 测试要求程序中从每个变量的每个定义到这个定义的每个使用之间的每个 du path 在测试中至少被经过一次。

- All-use 测试

All-use 测试要求程序中从每个变量的每个定义到这个定义的每个使用之间,至少有一条定义清晰的路径在测试中被经过一次。

- All-p-use/some-c-use 测试

All-p-use/some-c-use 测试要求程序中从每个变量的每个定义到这个定义的每个 p-use (即定义在谓词中被使用)之间,至少有一条定义清晰的路径在测试中被经过一次。如果那个定义没有 p-use,把上面语句中的“每个 p-use”替换成“至少一个 c-use”(即,定义在计算中被使用)。

- All-c-use/some-p-use 测试

All-c-use/some-p-use 测试要求程序中从每个变量的每个定义到这个定义的每个 c-use 之间,至少有一条定义清晰的路径在测试中被经过一次。如果那个定义没有 c-use,把上面语句中的“每个 c-use”替换成“至少一个 p-use”。

- All-definition 测试

All-definition 要求程序中每个变量的每个定义,在测试期间至少有一条从该定义出发的 du path 被经过一次。

- All-p-use 测试

All-p-use 测试要求程序中从每个变量的每个定义到这个定义的每个 p-use 之间,至少有一条定义清晰的路径在测试中被经过一次。

- All-c-use 测试

All-c-use/some-p-use 测试要求程序中从每个变量的每个定义到这个定义的每个 c-use 之间,至少有一条定义清晰的路径在测试中被经过一次。

图 5-10 显示了基于源代码结构的测试用例选择标准之间的覆盖率关系。

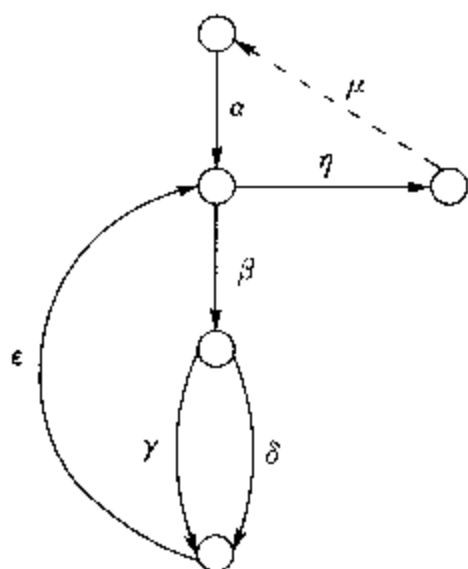


图 5-9 添加了出口点到入口点边的控制流图

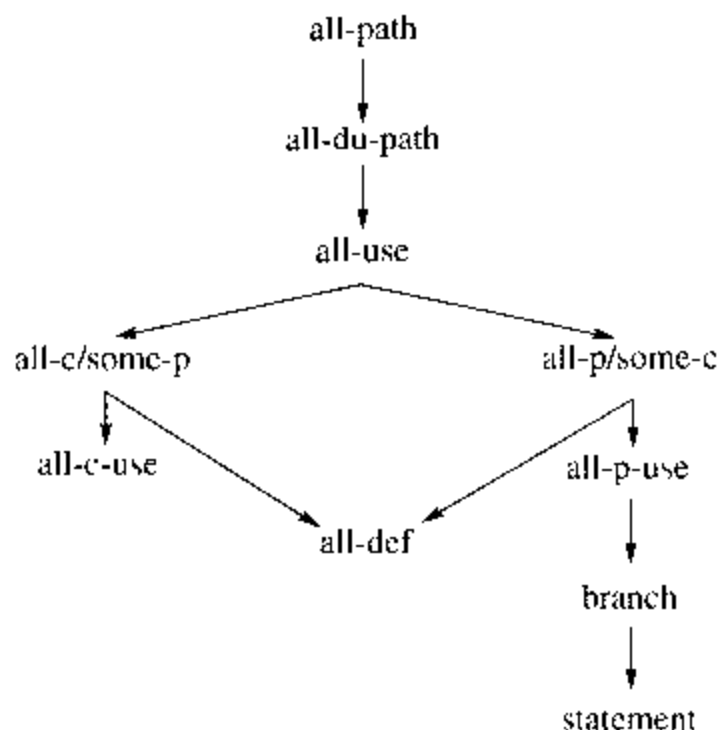


图 5-10 各测试用例选择标准之间的覆盖率关系

2. 基于编译器测试

Hamlet 描述了增加一个编译器来判断测试数据有效性的方法^{[59][60]}。输入输出数据作为对被编码的一个过程的注释，作为那个过程被计算功能的部分规格。然后过程对每个输入执行进行执行并且对输入值进行检查。测试被认为是充分的条件是如果在过程中的每个计算或逻辑表达式被测试确定；即，没有表达式可以被一个更简单的表达式替代并且还通过了测试。在这里，更简单被定义为一种方法，它只允许有限的许多置换。这样，随着过程被执行，根据传递给表达式的测试数据对每个可能的置换进行评价。那些评价和原始表达式不一样的置换被拒绝。同样评价相同但是最终产生失败的置换也被拒绝。

3. 数据流测试

数据流分析可以组成测试的基础，它探索变量被定义的点和它们被使用的点之间的关系来发现缺陷^{[77][78][79][80][81][82]}。通过检视不同定义使用对的覆盖率，数据流测试建立了一些必须的条件用于影响和部分传递。数据流测试背后的动机是如果测试数据不能执行这些不同的定义-使用对组合，那么它们是不完整的。很显然，一个在测试中从来没有使用过的不正确的定义是不会被测试发现的。同样，如果一个给定的不正确的位置使用了一个特殊的定义，但是那个组合在测试期间从没有被尝试过，那么缺陷是不会被检测到的。

数据流关系可以被静态^[82]或动态^[83]确定。由于不可达子路径的存在，一些关系可能不可实现。启发式方法可以根据数据流信息来产生测试数据。

常见的数据流异常包括：

- 引用一个未初始化的变量；
- 一个变量的死(无用)定义；
- 等待一个还没有被安排的进程；
- 安排一个与其自身相同的进程；

- 等待一个先前已经被中止了的进程；
- 引用一个在并行进程中被定义的变量；
- 引用一个值不确定的变量。

请看下面的例子：

```
1      Main: program;
2          declare integer x, y;
           /* x and y are global variables known throughout
           the main programs and all tasks * /
3          declare boolean flag;
4          T1: task;
5              write x;
6              wait for T3;
7          Close T1;
8          T2: task;
9              x = 5;
10             y = 6;
11         close T2;
12         T3: task;
13             read x;
14         close T3;
           /* end of declarations * /
15         schedule T1; /* first executable statement of Main * /
16         schedule T2;
17         read flag;
18         if flag then x = 8;
19         write x;
20         y = 9;
21         wait for T2;
22         if flag then y = 10;
23         write y;
24         wait for T2;
25         schedule T1;
26     close Main;
```

关于这段程序，可以获得如下数据流分析图，如图 5-11。

从图 5-11 中，可以分析出该程序大致可能出现的异常有：

- 变量 x 在语句第 5 行处的引用可能没有被定义，这是因为任务 T1 可能在任务 T2 开始前结束；
- 在任务 T2(第 10 行)以及主程序第 20 行发现的变量 y 的定义可能没有用，这是因为 y 会在其可能被引用前在第 22 行被重新定义；
- y 被两个可能并行执行的进程定义，并且这样在第 23 行的引用其值可能是无法确定的；
- 变量 x 在任务 T2(第 9 行)被赋了一个值，然而，同时又在主程序第 19 行处被引用；
- 存在这样一个可能性，即 T1 可能在第 25 行被并行安排，因为没有任何保证 T1 在最初被安排后会终结；

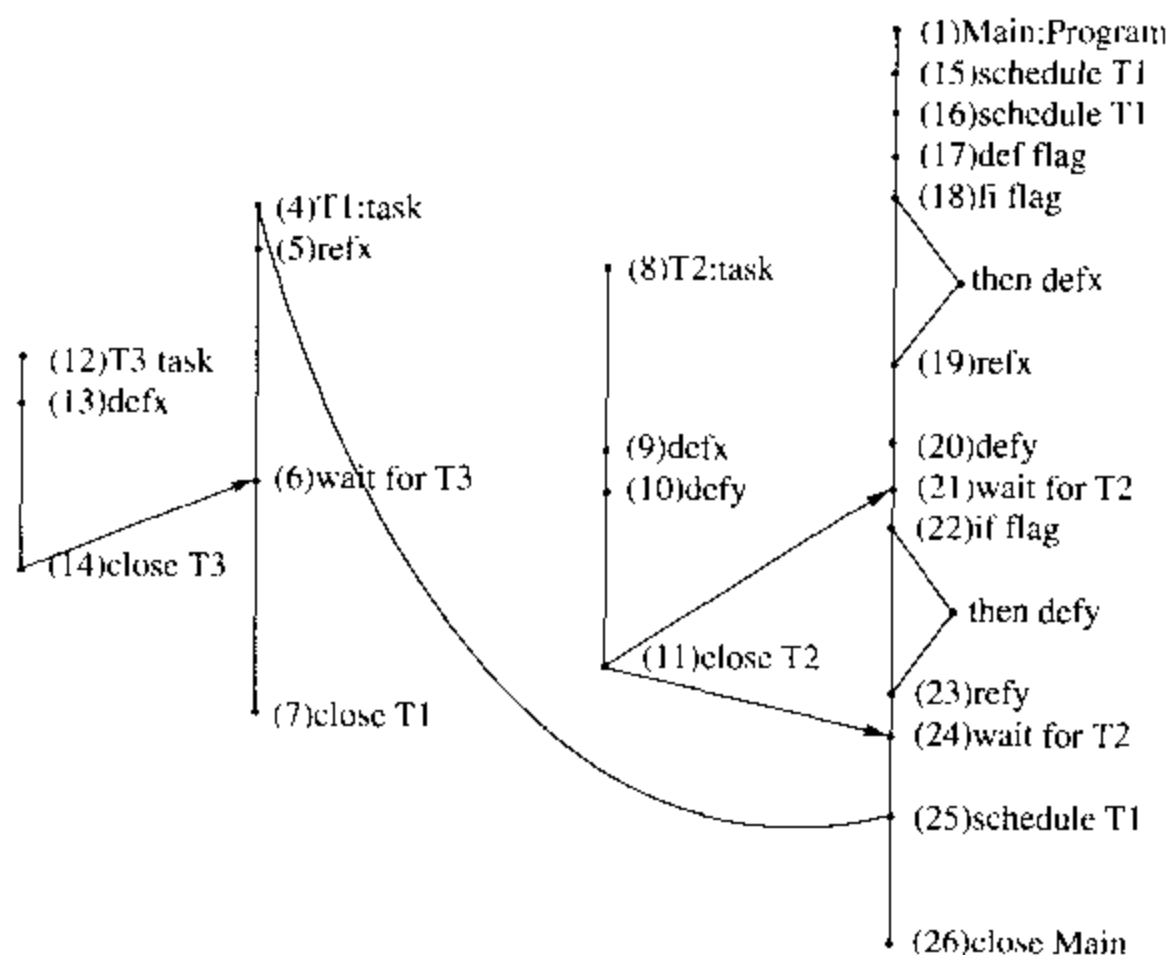


图 5-11 数据流分析图例子

- 第 24 行的等待是没有必要的，因为 T2 被保证在第 21 行终结，并且随后没有再被安排过；
- 第 6 行的等待永远不会被满足，因为 T3 从来没有被安排过。

4. 故障插入测试

有关故障插入的概念及相关知识请参考本书 4.5.1 节的内容。

香港中文大学 Michael Lyu 教授在故障插入方面进行了比较多的研究^[147]。根据他的研究，一个故障插入的策略可以用图 5-12 来表示，

其中的步骤可以分为：

- 故障插入测试前步骤

- (1) 软件构架分析
- (2) 主因分析
- (3) 测试用例选择
- (4) 测试计划

- 故障插入测试步骤

- (1) 故障/错误/失败 插入
- (2) 测试执行出发器
- (3) 观察行为

- 故障插入测试后步骤

- (1) 测试结果评价
- (2) 测试覆盖率评价

软件构架分析是一个正向的行为，它分析被测软件的结构、组件、容错机制等内容，

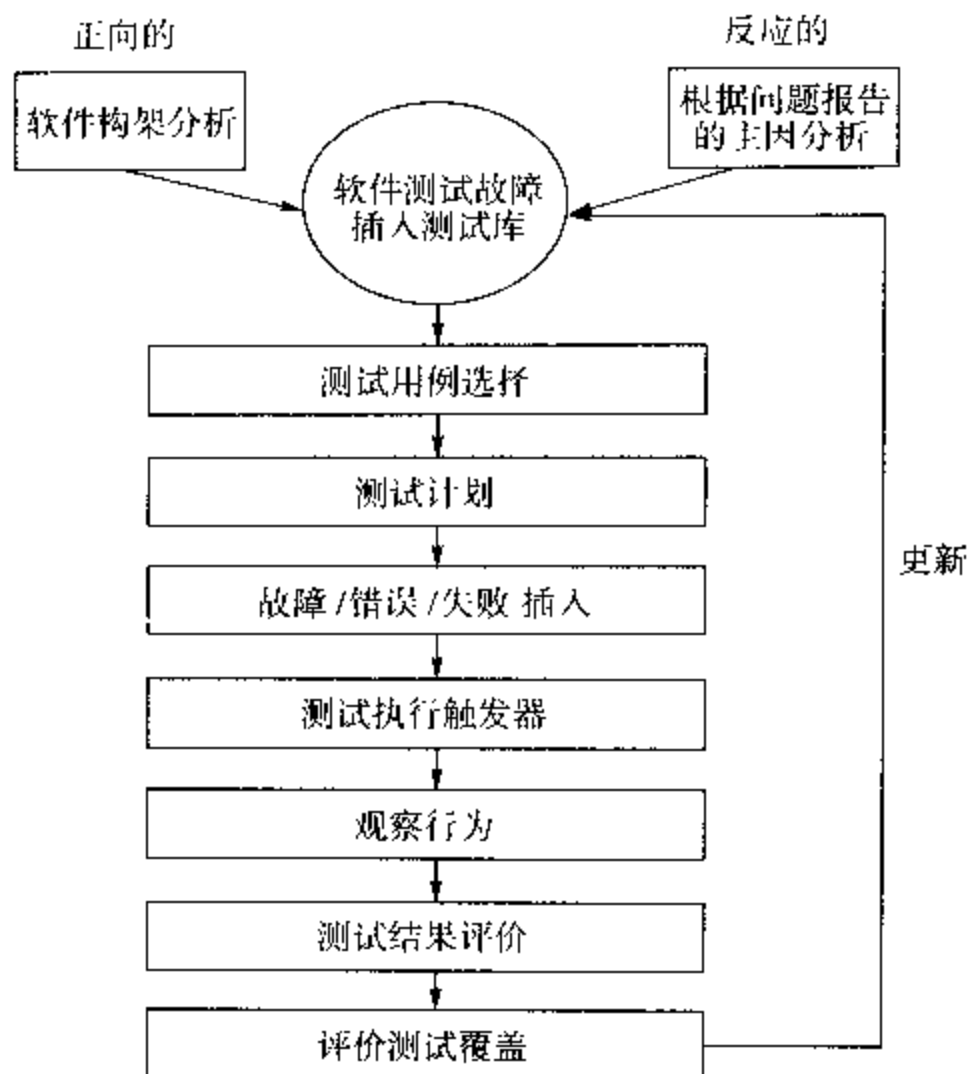


图 5-12 软件故障插入测试策略

具体如下：

- 业务模块分析(着重于有潜在风险的地方)

- (1) 新的和修改过的模块
- (2) 关键功能模块
- (3) 容易出错的软件模块
- (4) 测试覆盖率低的模块
- (5) 软件质量属性较差的模块

- 故障管理分析

- (1) 故障处理机制
- (2) 观测机制
- (3) 和故障管理的模块交互

- 故障管理/服务管理接口分析

- (1) 出错代码和它们的相关属性
- (2) 可能的事件次序

主因分析是一个反应性的事后的活动，它根据以前发现的问题进行主要原因分析，确认需要通过故障插入测试来验证的一些公共问题。

测试用例选择是一个关键步骤，可以通过以下方式来进行：

- 确认要被检查的属性和谓词；
- 确认要进行的观察行为；
- 为覆盖率目的而考虑的标准；

- 用于加强覆盖率;
- 用于在实际操作中模拟故障/错误;
- 用于分析直接影响客户业务的错误;
- 用于分析影响范围较大的故障/错误。

测试计划需要考虑:

- 测试脚本;
- 用于代码注入的软件补丁;
- 用于状态注入的适当工具。

故障/错误/失败插入考虑:

- 确认的位置;
- 使用的工具。

测试触发器考虑:

- 从用户输入的值;
- 内部或外部的事件;
- 消息。

观察行为主要观察系统在一个时间段内的反应。

测试结果评价主要考虑:

- 评价测试用例有效性;
- 评价系统的容错能力。

测试覆盖率评价主要考虑:

- 在故障库中故障、错误或失败类型的覆盖情况;
- 故障管理的功能覆盖情况;
- 故障管理的结构化覆盖情况;
- 在业务管理中的高危模块覆盖情况;
- 产生于软件故障树分析的覆盖情况。

更新 SFIT(Software Fault Insert Testing)测试用例库考虑:

- 构造一个库来收集故障、类型和属性;
- 包含一般和特殊的故障;
- 制定用于容错性测试的测试充分性标准。

5. 变体测试

变体测试(Mutation Testing)使用变体分析(参考本书4.1.5节内容)来判定测试数据的充分性。测试数据被认为是充分的条件是:(1)在每个变体与最初的程序功能相等;(2)在测试数据上,其计算结果不同于最初程序。测试数据的不充分意味着某些缺陷被引入到代码中并且没有被测试数据发现。

我们假定 P' 是程序 P 的一个变体。对于一个测试用例 t , 如果 P 和 P' 对于该用例有不同的结果, 我们称 t 从 P 中区分 P' 。如果 t 不能从 P 中区分 P' , 那么要么 P 在功能上等同于 P' , 要么 t 在揭示引入到 P' 中的错误上是没有效果的。这样一个测试方法可以被形式化成下面的形式。

对于一个实现功能 f 的程序 P ：

第一步：通过使用变体操作集合(参考本书4.1.5节内容)产生 P 的一个变体集合 Φ ；

第二步：在 Φ 中确认和删除所有和 P 等价的变体；

第三步：寻找一个能够完全区分 P 和 Φ 中每个变体的测试用例集合来测试-执行 P 和 Φ 中的每个元素。

上面3个步骤组成了一个 Φ 变体测试。如果对于所有 $t \in T$ 有 $P(t) = f(t)$ ，那么这个测试是成功的。一个成功的 Φ 变体测试意味着程序对在构造 Φ 时任何引入到 P 中的错误是免疫的。如果假定 P 是由一个能胜任的程序员编写的，那么可以得出一个结论，即 P 是正确的。进一步的内容可以参考^{[40][87]}。

变体测试基于两个假定^[61]：能胜任的程序员假定(Competent Programmer Hypothesis)和耦合效果假定(Coupling Effect Hypothesis)。能胜任程序员假定说的是一个胜任的程序员写的代码是接近正确的；如果当前的程序不是一个正确的程序，那么正确的程序可以通过一些对代码的直接句法变化得到。耦合效果假定说的是揭示简单错误的测试数据不会发现复杂的缺陷。这样，只有简单的变体需要被排除，并且多个变体的组合效果不需要被考虑^[61]。Gourlay和John S正式把能胜任程序员假定的特性作为测试集合可靠性的一个功能并且表明在此特性下，假设可以不坚持^[84]。耦合效果的经验证明已经被很多人尝试过^{[87][61][86]}，但是理论分析显示它可能在某些情况下是正确的，但是不具有普遍性^{[84][85]}。

程序变体测试根据变异本身可以分为程序强变体测试(Strong Mutation Testing)和程序弱变体测试(Weak Mutation Testing)^{[75][111]}。

- 强变体测试

强变体测试就是平常所说的变体测试。

- 弱变体测试

弱变体的目的是要查出某一类型错误。其主要思想是：

假设 P 是一个程序， C 是 P 的简单组成部分。若有一个变体变换作用于 C 而生成 C' ，如果 P' 是含有 C' 的 P 的一个变体，则在弱变体方法中，要求存在测试数据使得 P 在该数据下运行时， C 被执行到，且至少存在一次执行使得 C 产生的值与 C' 不同。使用弱变体测试，变体和最初程序的状态可以在变体执行后尽快地被比较，而不必等到整个程序被运行结束。

在弱变异的实现中，关键问题是确定程序 P 的组成部分集合及其相关的变换。Howden提出了5种最基本的程序组成部分：变量引用、变量定义、算术表达式、关系表达式和布尔表达式。其中前两个部分可以包含在后3个部分中。算术表达式可以包含在关系表达式中，而关系表达式又可以包含在布尔表达式中^{[1][75]}。

弱变体测试比强变体测试成本要低很多，但效果几乎是相同的^[111]。

5.3 面向错误的测试

由于在编程过程中会有潜在错误的产生，因此测试是必需的。关注于在编程过程中错误出现或消失的评价的技术被称为面向错误的技术。

5.3.1 基于错误的测试

基于错误的测试(Error-Based Testing)是寻找和证明某个错误已经出现在程序过程中了^[88]。它可以通过程序员犯错误的历史、软件复杂度的度量、易于出错的句法结构的知识,或者错误猜测来驱动^[2]。

基于错误的测试开始于编程过程,确定在哪个过程中潜在的错误,然后寻找那些错误如何反应在故障上。Howden已经把错误分为两类:抽象和分解,并且他已经开发了用于表述那些分类的特殊技术^{[89][90]}。

1. 错误猜测法

错误猜测(Error Guessing)法是基于经验和其他一些测试技术,如边界值测试的一种方法。在经验的基础上,测试设计者猜测错误的类型以及在特定的软件中错误发生的位置并设计测试用例去发现它们。例如,如果所有的资源需要动态申请,一个发现错误的好地方就是资源释放的地方。是否所有的资源都被正确地释放了,或者在软件执行过程中丢失了?对于一个有经验的工程师,错误猜测法可能是一个最有效的设计测试用例发现问题的方法。一个好的错误猜测可能发现被别的测试方法很容易遗漏的错误。但反过来,错误猜测也可能是白白浪费时间。

为了最大限度地利用有效的经验并逐步丰富测试用例设计技术,建立一个错误类型的列表不失为一种好方法。这个列表可以帮助“猜测”程序中的错误会在哪里出现。这个列表应通过早期的程序测试经验不断维护,以帮助改进错误猜测的有效性。

我们使用5.1.2中第5点使用到的例子来说明错误猜测法的使用,规格描述如下。

写一段程序来重新格式化文本,具体如下。

给定一个文本,以ENDOFTEXT字符结尾,并且包含的字使用BLANK或者NEWLINE字符隔开,根据下面规则把该文本转成一行接一行的格式:

- 行在文本有BLANK或NEWLINE处中断;
- 行尽可能长;
- 没有行可以有超过MAXPOS长度的字符。

这段规格会使一个有经验的程序员怀疑程序能否在下面这些情况下正确工作(因此可以从这些情况中选择测试用例):

- 输入文本长度是零;
- 文本包含一个非常长的字(超过MAXPOS);
- 文本除了BLANK和NEWLINE之外没有任何别的字符;
- 一个文本有一个空行;
- 字被两个或更多个NEWLINE或BLANK分隔;
- BLANK为行的开头或结束字符;
- 文本包含数字或特殊字符;
- 文本包含不可打印字符;
- MAXPOS被设置为一个超过系统默认行长度的数字。

B. W. Kernighan 等人在编程类型方面的研究可能会帮助你积累错误猜测方面的经验^[109]。

5.3.2 基于故障的测试

基于故障的测试(Fault-Based testing)瞄准的是证明某个规定的故障不存在于代码中。基于故障的测试在范围和广度方面是不同的。使用局部范围的方法证明一个故障对计算有局部影响；很可能这个影响不会产生一个程序失败。使用全局范围的方法证明一个故障会引起一个程序失败。广度决定于技术是处理一个有限还是无限故障集。范围和广度是正交的。面向影响和面向传递的技术可以根据故障进行分类。面向影响的技术是局部范围的。

Morell 和 Larry J 根据符号执行定义了一个基于故障的方法，它允许从全局失败的迹象中排除无穷多的故障。符号故障(Symbolic Faults)被插入到代码中，然后根据实际的和符号的数据执行。程序输出是一个根据符号故障的表达式。这样它反应了在一个给定的位置一个故障是如何影响程序的输出的。这个表达式可以被用于确定实际的故障。

5.3.3 基于风险的测试

什么是基于风险的测试(Risk-Based Testing)?

- 做一个风险的优先级列表
- 进行测试来探询每个风险
- 随着老的风险消失，新的风险产生，调整测试工作重点到新的风险上

这就是基于风险的测试。风险是可能发生的问题。一个风险的大小与问题产生的可能性和影响力有着密切的联系。和问题相关的风险越高，问题就越有可能产生，并且当问题产生后的影响也越大。这就是为什么要进行基于风险的测试的动机。

如果你对一个高可靠性产品的测试负责，那么你可能想要使用一种严格的风险分析格式。这种方法使用了统计模型和/或分析灾难和失败模型^{[113][114]}。在此我们介绍一种启发式的风险分析方法。

启发式分析方法可以追溯到 George Polya 的经典著作“*How to Solve It*”。关于该方法在软件需求开发方面的应用可以参考附录参考资料中的^[116]。在进行基于风险的测试时，还得考虑系统中有哪些风险，从经验来讲，对于任何系统来说都存在的一般性风险包括：

- 程序的复杂度
- 全新的软件或模块
- 变更的软件或模块
- 对上游软件的依赖，发生错误会导致雪崩效应的模块
- 对下游软件的依赖，对错误特别敏感的模块
- 关键软件或模块
- 对需求符合的完整性
- 经常使用的模块
- 对业务特别重要的模块

- 第三方软件或模块
- 在空间或时间上是分布式的，但是需要作为一个整体才能工作的代码
- 已知的问题较多的软件或模块
- 最近发生过故障的软件或模块

从风险的质量分类来看包括：

- 能力(Capability)
- 可靠性(Reliability)
- 可用性(Usability)
- 性能(Performance)
- 可安装性(Installability)
- 兼容性(Compatibility)
- 支持性(Supportability)
- 可测试性(Testability)
- 可维护性(Maintainability)
- 可移植性(Portability)
- 区域性(Localizability)

Cem Kanar 在其《Testing Computer Software》一书的附录 A 中给出了一个比较广泛的风险分类目录^[5]，有兴趣的读者可以参考一下。下面给出关于进行基于风险测试的一个基本步骤：

- 决定要分析什么组件或功能
- 确定关心的范围，可以使用“高风险”，“一般风险”和“低风险”来划分
- 收集想要分析的对象的信息(或有这些信息的人员)
- 观察每个列表上的每个风险区域，根据手头上的资料确定其重要性
- 记录任何不清楚的，影响分析风险能力的事情
- 再次分开检查所有风险

5.3.4 可能的正确性

可能的正确性(Probable Correctness)被定义为在一个被测程序中没有故障的可能性^[91]。早期涉及到这个概念的资料可以在附录中参考资料的^[93]和^[94]中找到，其中一些特定类型的功能存在一些成员，这些成员可以通过一个有限的测试集同其他的成员区分开来。这样，每个成功的执行增加了实现功能正确性的信心。

5.4 混合测试技术

很明显，没有哪个测试分析技术是足够的，一些专家已经研究了一些综合多种技术的方法。这种集成的技术被称为混合测试技术(Hybrid Testing Techniques)。这些不是仅仅的并行使用这些不同的技术；它们的特点是小心地综合不同方法的最好特性到一个新的方法

中去。

在等价类分析中,测试数据被选择用于保证规格和代码的同时覆盖^[65]。一个可操作的规格语言已经被设计,它使得规格的覆盖率的一个结构化度量成为可能。输入空间被划分成一组域,它通过程序的路径域和规格的路径域交叉乘积组成。测试数据从每个非空的分类中选择,保证规格和代码的同时覆盖。正确性技术的证明也可以用于这些交叉乘积域。

测试系统 DAISTS 自动化了这个技术用于抽象数据类型的代数规格,但是它不能包含任何正确性证明的想法^[47]。此外,在 DAISTS 中强调的是测试数据的评价而不是产生。Goodenough 等人给出了一个不是很形式化的,关于选择测试数据的集成框架,它基于在程序过程中错误源的分析。

5.5 本章小结

本章从面向规格、面向实现和面向错误 3 个维度介绍了业界许多常见的测试分析方法。这些方法各有特点和应用范围。许多方法之间也存在着彼此的联系,有些方法也很难确切归类,也不是每种方法都能适合任何系统,也不是只用一种方法就能测试你的系统,那将是不完备的。但是,无论从时间上还是成本上考虑,试图用所有方法来测试也是不可取的。因此,在选择合适的测试分析方法时,考虑到从需要测试对象的特点、测试的重点,以及历史经验等因素进行综合分析。一般来说,规范导出法、判定表、因果图、面向结构的测试(基于结构化覆盖率),错误猜测法和基于风险的测试都是比较容易上手的,且成本也比较容易控制。面向传递的测试方法在测试有效性方面比较突出,但需要投入比较大的成本。

第6章 单元测试

从传统的动态测试概念来讲，单元测试是最早开始的测试，属白盒测试范畴。可以把单元测试认为是开发过程中的一个阶段，但从现在的观点来看，这个阶段在详细设计时就可以开始了。更好的理解是，可以把它看作是一个活动，该活动起始于详细设计，并且会一直延续到项目的终结。之所以这么认为，是因为项目在生命周期后期，还会进行更改（修正缺陷，加入新功能等）。进行这些活动后，都应当再进行单元测试。本章将重点讨论单元测试的概念。本章的组织结构包括：

1. 对单元测试概念的阐述。
2. 为什么要进行单元测试？
3. 单元测试的环境，包括什么是桩，什么是驱动。
4. 如何进行单元测试，考虑测试执行策略，测试分析内容，用例设计方法及测试过程。
5. 常用的单元测试工具。
6. 进行单元测试的原则。
7. 目前在国内软件企业中存在的单元测试问题。
8. 对本章进行一个简单的小结。

自从软件危机发生以来，软件质量和可靠性被业界看得越来越重要。软件的质量和可靠性经常是一个企业试图开发新产品或提供新服务最弱的环节。在过去的几十年中，随着越来越多的人在开发过程中采用了设计方法论和使用 CASE 工具，软件质量和可靠性问题越来越受到重视。大多数软件设计人员都接受了这方面的培训，并且在这些正规的软件设计方法使用中取得了很多经验^[6]。

不幸的是，软件测试并没有得到同样的重视。很多使用这些软件设计方法的开发活动并没有使软件质量和可靠性得到控制。修改最初的软件开发活动遗留的 Bug 一般要在软件维护费用中占到 50% 的比例，这是不正常的。这些 Bug 应该在有效的软件测试过程中被排除。

测试是软件工程中一个非常广阔的范畴。测试的种类有很多，如：白盒测试、黑盒测试、单元测试、集成测试、系统测试、负荷测试等。单元测试可以说是最早的基于代码运行的测试，是在软件开发过程中要进行的最低级别的测试活动，也是非常重要的测试。

6.1 什么是单元测试

单元测试(Unit Testing)是对软件基本组成单元进行的测试，这里的基本单元不一定是 指一个具体的函数(function 或 procedure)或一个类的方法(method)。“单元”具有一些基

本属性，如：明确的功能、规格定义，与其他部分明确的接口定义等，可清晰地与同一程序的其他单元划分开来。在具体实现时，也可能对应多个程序文件中的一组函数。

在一种传统的结构化编程语言中，比如 C，要进行测试的单元一般是函数或子过程。在类似 C++ 这样的面向对象的语言中，要进行测试的基本单元是类或类的方法。对 Ada 语言来说，开发人员可以选择独立的过程和函数，或在 Ada 包级别上进行单元测试。单元测试的原则同样被扩展到第四代语言(4GL)的开发中，在这里基本单元被典型地划分为一个菜单或显示界面。

在纯 C 语言的代码中，我们一般认为一个函数就是一个单元，这主要是为了避免开发人员和测试人员陷入不必要的单元争论当中去。同时也可以避免歧义。

在实际的单元测试执行过程中，有时会遇到某个函数 A 只被函数 B 调用，并且函数 A 和函数 B 的代码总和在一定的范围之内，则可以考虑把 A 和 B 合并作为一个单元进行测试，但这些原则必须在单元测试计划或单元测试方案中明确说明。

6.1.1 单元测试的目的

测试的目的有很多，Grenford J. Myers 认为软件测试目的在于发现错误^[2]；一个好的测试用例在于发现至今未发现的错误；一个成功的测试是发现了至今未发现的错误的测试。因此，很多开发人员习惯性地认为单元测试是为了检测代码中的错误。这个观点不能说是错，但是不严谨。从现代软件工程的角度来看，软件的开发质量是从过程上进行保证的。如果说在进行编码前已经能很大程度上保证详细设计的质量，那么做单元测试时就不仅仅要检测代码的错误，而需要测试代码是否是根据详细设计进行的。因此，单元测试的主要目的有：

- 验证代码是与设计相符合的；
- 跟踪需求和设计的实现；
- 发现设计和需求中存在的错误；
- 发现在编码过程中引入的错误。

6.1.2 单元测试和集成测试的区别

单元测试与集成测试相比，测试对象有所区别。集成测试的被测对象是在概要设计中规划的模块及这些模块间的组合。这里，不同模块往往是分配给不同的某个(或某组)程序人员开发。单元测试的测试对象是这些模块下实现具体功能的单元，一般是对应详细设计中所描述的设计单位。

集成测试关注的是模块间的接口，接口之间的数据传递关系，单元组合后是否实现预计的功能等。集成测试组装的对象比单元测试的对象级别要高。如果说单元测试的对象是一个一个函数，那么集成测试组装的对象可能是一个一个接口函数(公共函数)。对于那些非接口函数则直接挂在接口函数上集成到系统中去。

从现代的测试来看，单元测试和集成测试之间的界限也变得模糊起来。单元测试方法

中也引入了集成概念，如：为了减少桩模块的设计，单元测试采用由低向上的测试方法。其目的是为了发现开发中的错误，提高产品的质量，而不是去追究单元测试和集成测试到底有多严格的区别。如果这样，那么我们就走入歧途了。

6.1.3 单元测试和系统测试的区别

单元测试和系统测试之间的区别比较明显。一般来说单元测试属于白盒测试，关注的是单元的具体实现、内部的逻辑结构、数据流向等；单元测试使问题及早暴露，也便于问题的定位解决。单元测试属于早期测试，因而错误发现后就能明确知道是由某一单元产生的；单元测试允许多个被测单元的测试工作同时开展。

系统测试则属于黑盒测试，是站在用户的角度上来看待系统，对系统进行测试，证明系统是否已经满足了用户的需要。其测试是基于需求规格说明书。系统测试是一种后期测试，错误发现后的定位工作比较困难。

关于白盒测试和黑盒测试的概念请参考本书第2章的内容。

6.2 为什么要进行单元测试

在开发过程中使用了静态测试手段(如检视，走读和评审等)，后期又有集成测试和系统测试，那么是否还有必要做单元测试呢？毕竟单元测试的工作量是相当庞大的。IPL给出了在开发人员中常见问题的一个解答^[119]。

问题一、单元测试浪费了太多时间

一旦编码完成，开发人员总是会迫切希望进行软件的集成工作，这样他们就能够看到实际的系统开始启动工作了。这在外表上看来是一项明显的进步，而像单元测试这样的活动也许会被看作是通往这个阶段点道路上的障碍，推迟了对整个系统进行集成这种真正有意思的工作启动时间。

在这种开发步骤中，真实意义上的进步被外表上的进步取代了。系统能够正常工作的可能性是很小的，更多的情况是充满了各式各样的 Bug。在实践中，这样一种开发步骤常常会导致这样的结果：软件甚至无法运行。更进一步的结果是大量的时间将被花费在跟踪那些包含在独立单元中的简单的 Bug 上面，在个别情况下，这些 Bug 也许是琐碎和微不足道的，但是总的来说，它们会导致在软件集成为一个系统时增加额外的工期，而且当这个系统投入使用时也无法确保它能够可靠运行。

在实践工作中，进行了完整计划的单元测试和编写实际的代码所花费的精力大致上是相同的。一旦完成了这些单元测试工作，很多 Bug 将被纠正，在确信他们手头拥有稳定可靠的部件的情况下，开发人员能够进行更高效的系统集成工作，这才是真实意义上的进步。所以说完整计划下的单元测试是对时间的更高效的利用，而调试人员的不受控和散漫的工作方式只会花费更多的时间和取得很少的成果。

使用 AdaTEST 和 Cantata 这样的支持工具可以使单元测试更加简单和有效。但这不是

必须的，单元测试即使是在没有工具支持的情况下也是一项非常有意义的活动。

问题二、单元测试仅仅是证明这些代码做了什么

这是那些没有首先为每个单元编写一个详细规格说明而直接跳到编码阶段的开发人员提出的一条普遍的抱怨，当编码完成以后并且面临代码测试任务时，他们就阅读这些代码并找出它实际上做了什么，把他们的测试工作基于已经写好的代码的基础上。当然，他们无法证明任何事情。所有的这些测试工作能够表明的事情就是编译器工作正常。是的，他们也许能够抓住(希望能够)罕见的编译器 Bug，但是他们能够做的仅仅是这些。

如果他们首先写好一个详细的规格说明，测试能够以规格说明为基础。代码就能够针对它的规格说明，而不是针对自身进行测试。这样的测试仍然能够抓住编译器的 Bug，同时也能找到更多的编码错误，甚至是一些规格说明中的错误。好的规格说明可以使测试的质量更高，所以最后的结论是高质量的测试需要高质量的规格说明。

在实践中会出现这样的情况：一个开发人员要面对测试一个单元时只给出单元的代码而没有规格说明这样吃力不讨好的任务。怎样做才会有更多的收获，而不仅仅是发现编译器的 Bug？第一步是理解这个单元原本要做什么，而不是它实际上做了什么。比较有效的方法是倒推出一个概要的规格说明。这个过程的主要输入条件是阅读那些程序代码和注释，主要针对这个单元，及调用它和被它调用的相关代码。画出流程图是非常有益的，此外，还可以用手工或使用某种工具。可以组织对这个概要规格说明的走读，以确保对这个单元的说明没有基本的错误，有了这种最小程度的代码深层说明，就可以用它来设计单元测试了。

问题三、我是个很棒的程序员，我是否可以不进行单元测试

在每个开发组织中都至少有一个这样的开发人员，他非常擅于编程，他们开发的软件总是在第一时间就可以正常运行，因此不需要进行测试。你是否经常听到这样的借口？

在真实世界中，每个人都会犯错误。即使某个开发人员可以抱着这种态度在很少的一些简单的程序中应付过去。但真正的软件系统是非常复杂的，它不可以寄希望于没有进行广泛的测试和 Bug 修改过程就可以正常工作。

Humphry 在其 PSP 的研究中发现^[122]：

- 一般软件工程师平均缺陷引入率在 100 个/千行代码(包括编译错误)；
- 受过 PSP 训练的工程师缺陷引入率在 50 个/千行代码；
- 工程师在编写代码的时候，一般每小时引入 6~8 个缺陷；
- 在设计阶段，一般每小时引入 1~3 个缺陷。

编码不是一个可以一次性通过的过程。在真实世界中，软件产品必须进行维护以对操作需求的改变作出反应，并且要对最初的开发工作遗留下来的 Bug 进行修改。你指望那些原始作者进行修改吗？那些制造出这些未经测试的原始代码的资深专家们还会继续在其他地方制造这样的代码。在开发人员做出修改后进行可重复的单元测试可以避免产生那些令人不快的副作用。

附录 C 给出了一个常见的编码错误分析，这对保证编码质量和单元测试都有比较好的指导意义。

问题四、不管怎样，集成测试将会抓住所有的 Bug

我们已经在前面的讨论中从一个侧面对这个问题进行了部分阐述。这个论点不成立的原因在于规模越大的代码集成意味着复杂性越高。如果软件的单元没有事先进行测试，开发人员很可能会花费大量的时间仅仅是为了使软件能够运行，而任何实际的测试方案都无法执行。

一旦软件可以运行了，开发人员又要面对这样的问题：在考虑软件全局复杂性的前提下对每个单元进行全面的测试。这是一件非常困难的事情，甚至在创造一种单元调用测试的条件时，要全面地考虑单元被调用时的各种入口参数。在软件集成阶段，对单元功能全面测试的复杂程度远远超过独立进行的单元测试过程。

最后的结果是测试将无法达到它所应该有的全面性。一些缺陷将被遗漏，并且很多 Bug 将被忽略过去。

让我们类比一下，假设我们要清洗一台已经完全装配好的食物加工机器！无论你喷了多少水和清洁剂，一些食物的小碎片还是会粘在机器的死角位置，只有任其腐烂并等待以后再想办法。但我们换个角度想想，如果这台机器是拆开的，这些死角也许就不存在或者更容易接触到了，并且每一部分都可以毫不费力地进行清洗。

问题五、它的成本效率不高

一个特定的开发组织或软件应用系统的测试水平取决于对那些未发现的 Bug 的潜在后果的重视程度。这种后果的严重程度可以从小小的不便到发生多次死机的情况。这种后果可能常常会被软件开发人员忽视（但是用户可不会这样），这种情况会长期地损害这些向用户提交带有 Bug 软件的开发组织的信誉，并且会对未来市场产生负面的影响。相反地，一个可靠的软件系统的良好声誉将有助于一个开发组织获取未来的市场。

很多研究成果表明，无论何时做出修改都要进行完整的回归测试，在生命周期中尽早地对软件产品进行测试将使效率和质量得到最好的保证。Bug 发现得越晚，修改它所需的费用就越高，因此从经济角度来看，应该尽可能早地查找和修改 Bug。在修改费用变得过高之前，单元测试是一个在早期抓住 Bug 的机会。

相比后阶段的测试，单元测试的创建更简单，维护更容易，并且可以更方便地进行重复。从全程的费用来考虑，相比起那些复杂且旷日持久的集成测试，或是不稳定的软件系统来说，单元测试所需的费用是很低的。

Capers Jones 和 McGraw-Hill 给出了在一个功能点上准备测试、执行测试和修正缺陷所需要花费的时间度量^[123]，见图 6-1。从图中可以发现单元测试的成本效率比集成测试、系统测试和其他一些测试都要高。但这并不意味着就可以不做后期阶段的测试，它们仍旧是必须的，因为不同级别的测试所关注的重点不一样，另一方面，穷尽的测试是不可能的，系统总会有缺陷遗留在里面。

6.3 单元测试环境

这里必须先解释桩 (Stub) 和驱动 (Driver) 的概念。因为单元本身不是一个独立的程

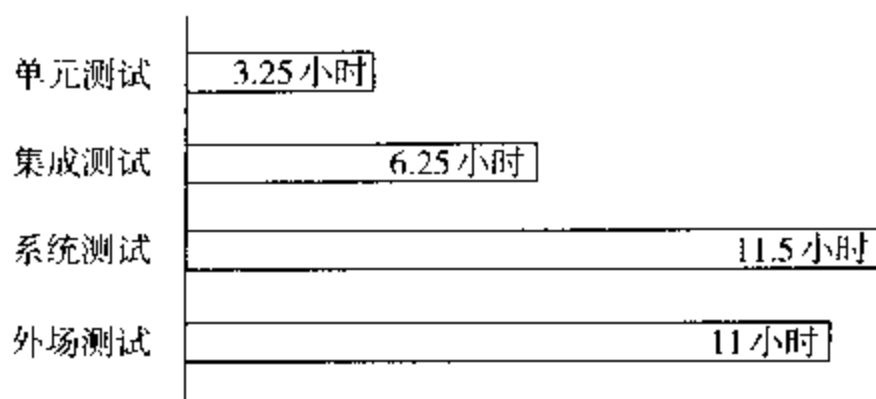


图 6-1 各种测试的时间效率

序，一个完整的可运行的软件系统并没有构成，所以必须为每个单元测试开发驱动模块和桩模块^[2]。在绝大多数应用中，驱动模块只是一个接收测试数据，并把数据传送给(要测试的)模块，然后打印相关结果的“主程序”。毫无错误的程序“桩模块”的功能是替代那些隶属于本模块(被调用)的模块，这种程序桩模块可能要使用子模块的接口，才能做一些少量的数据操作，并验证打印入口处的信息，然后返回。

驱动模块和桩模块都是额外的开销。也就是说，两种都属于必须开发但又不能和最终软件一起提交的软件。如果驱动模块和桩模块很简单，那么额外开销相对来讲是很低的。不幸的是，许多模块使用“简单”的额外软件是不能进行足够的单元测试的。在此情况下，完整的测试要推迟到集成测试阶段时完成。

构造单元的测试环境的主要工作有：

- 构造最小运行调度系统，即驱动模块，用以模拟被测模块的上一级模块；
- 模拟实现单元接口，即单元函数需调用的其他函数接口，即桩模块；
- 模拟生成测试数据或状态，为单元运行准备动态环境。

单元测试环境如图 6-2 所示。

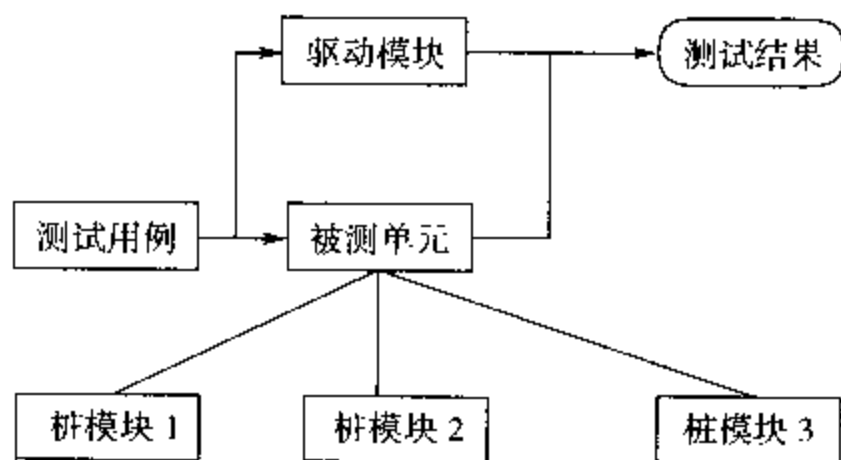


图 6-2 单元测试环境

测试环境除了要能使被测的对象运转起来之外，还应考虑对测试过程的支持。例如，对测试结果的保留，对测试覆盖率的记录等。对测试覆盖率的记录，可以通过多种方式，例如通过工具对被测函数插装，在测试完后再进行检查得出覆盖率数据等。当然，也可以直接在构造测试环境时就进行这方面的考虑。

测试的结论通过执行被测试的单元，根据导出的结果来得出，结论的有效性直接跟测试环境下模拟目标环境下(程序)执行的精确性相关。为保证在估计这些特征时所有的环境因素均被考虑，必须小心地设计桩和驱动。例如，所有的隐性输入必须被考虑(即，系

统时钟、文件状态、单元加载地点), 另外, 实际环境的代表物(即, 相同的编译器、加载者、操作系统、计算机、输入分布)也是测试环境必须考虑的。

6.4 单元测试策略

在如何选择单元测试策略时, 可以考虑3种方式: 由顶向下的单元测试策略(Top Down Unit Testing), 由底向上的单元测试策略(Bottom Up Unit Testing)和孤立的单元测试策略(Isolation Unit Testing)^[120]。

6.4.1 【策略一】由顶向下的单元测试策略

方法: 先对最顶层的单元进行测试, 把顶层所调用的单元做成桩模块。其次对第二层进行测试, 使用上面已测试的单元做驱动模块。如此类推直到测试完所有模块。

优点: 在集成测试前提供系统早期的集成途径。由于详细设计一般都是由顶到下进行设计的, 这样由顶向下的单元测试策略在执行上同详细设计的顺序一致。该测试方法可以和详细设计及编码进行重叠操作。

缺点: 单元测试被桩模块控制, 随着单元一个一个被测试, 测试过程将变得越来越复杂, 并且开发和维护的成本将增加。测试层次越到下层, 结构覆盖率就越难达到。同时任何一个单元的修改将影响到其下层调用的所有单元都被重新测试。低层单元的测试须等待顶层单元测试完毕后才能进行, 并行性不好, 测试周期将延长。

总结: 该策略比基于孤立单元测试的成本要高很多。不是单元测试的一个好的选择。但是如果单元都已经被独立测试过了, 可以使用该方法。

6.4.2 【策略二】由低到上的单元测试策略

方法: 先对模块调用层次图上最低层的模块进行单元测试, 模拟调用该模块的模块做驱动模块。然后再对上面一层做单元测试, 用下面已被测试过的模块做桩模块。以此类推, 直到测试完所有模块。

优点: 在集成测试前提供系统早期的集成途径。不需要桩模块。测试用例可以直接从功能设计中获取, 而不必从结构设计中获取。该方法在详细设计文档缺乏结构细节时变得有用。

缺点: 随着单元一个一个被测试, 测试过程将变得越来越复杂, 开发和维护的成本将增加。并且测试层次越到顶层, 结构覆盖率就越难达到。同时任何一个单元的修改将影响到直接或间接调用该单元的所有上层单元被重新测试。顶层单元的测试需等待低层单元测试完毕后才能进行, 并行性不好, 测试周期将延长。并且第一个被测试的单元一般都是最后一个被设计的单元, 单元测试不能和详细设计、编码进行重叠。

总结: 该策略是一个比较合理的单元测试策略, 尤其当需要考虑到对象或复用时。但由低向上的单元测试是面向功能的测试, 而不是面向结构的测试。这对于需要获得高覆盖

率的测试目标来说是相当困难的。并且该方法同紧凑的开发时间表相冲突。

6.4.3 【策略三】孤立测试

方法：不考虑每个模块与其他模块之间的关系，为每个模块设计桩模块和驱动模块。每个模块进行独立的单元测试。

优点：该方法是最简单，最容易操作的。可以达到高的结构覆盖率。由于一次只需要测试一个单元，其驱动模块比由低向上策略的驱动模块设计简单，其桩模块比由顶向下策略的桩模块设计简单。由于各模块不存在依赖性，所以单元测试可以并行进行，该方法对通过增加人员来缩短开发时间非常有效。该方法是纯粹的单元测试，上面两种策略是单元测试同集成阶段的混合。

缺点：不提供一种系统早期的集成途径(这不一定真的是缺点)。需要结构设计信息，使用到桩模块和驱动模块。

总结：该方法是最好的单元测试策略。如果辅助以集成测试策略，将可以缩短整个软件开发周期。

6.4.4 综合测试

在单元测试过程中，桩模块的工作量是相当庞大的。如何有效地减少这方面的工作量，可以考虑综合策略二和策略三这两种方法。例如，下面一个小程序：

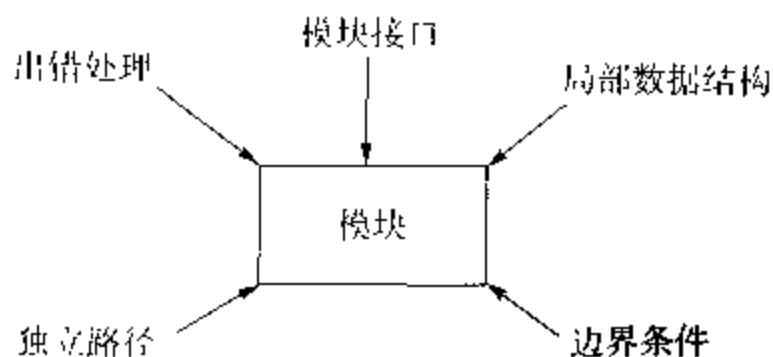
```
void funcA(int a, int b)
{
    if (max(a, b) < 0)
    {
        printf ("All input values are negative numbers! \n");
    }
    else
    {
        printf ("At least one of the input values is a zero or a natural number! \n");
    }
}

int max (int a, int b)
{
    if (a >= b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

为了减轻桩模块工作量，我们采用由底向上的测试策略，因此先对 max 函数进行了单元测试，然后直接使用 max 做桩来测试 funcA 函数。但根据策略二的方法，我们在测试 funcA 的时候是把 funcA 和 max 代码作为一个整体进行测试的，因此要求测试用例能够同时覆盖这两个函数。在此借鉴孤立测试策略，尽管我们直接使用了 max，但仍把它理解为桩，因此在设计用例时，我们不关注 max 本身怎么执行，而只关注该桩要返回一个小于零和大于等于零的值，以验证 funcA 能否在这两种情况下输出需要的信息。同时在考虑覆盖率时，也只需考虑 funcA 的覆盖率，而不必考虑 max 的覆盖率。

6.5 单元测试分析

在进行单元测试时，测试者要依据详细设计说明书和源程序清单，了解模块的 I/O 条件和模块的逻辑结构。一般可以从以下 5 个方面进行考虑^[2]，如图 6-3 所示。



6.5.1 模块接口

如果数据不能正确地输入和输出，就谈不上进行其他测试。因此，对模块接口需要如下的测试项目：

- 调用所测模块时的输入参数与模块的形式参数在个数、属性、顺序上是否匹配；
- 所测模块调用了子模块时，它输入给子模块的参数与子模块中的形式参数在个数、属性、顺序上是否匹配；
- 是否修改了只做输入用的形式参数；
- 输出给标准函数的参数在个数、属性、顺序上是否正确；
- 全局变量的定义在每个模块中是否一致；
- 约束条件是否通过形式参数来传送。

6.5.2 局部数据结构

模块的局部数据结构是最常见的错误来源。应设计测试用例以检查以下各种错误：

- 检查不正确或不一致的数据类型说明；
- 使用尚未赋值或尚未初始化的变量；
- 错误的初始值或错误的默认值；

- 变量名拼写错误或书写错误；
- 不一致的数据类型。

6.5.3 独立路径

对基本执行路径和循环进行测试会发现大量的错误。设计测试用例查找由于错误的计算，不正确的比较或不正常的控制流而导致的错误。

常见的不正确计算有：

- 运算的优先次序不正确或误解了运算的优先次序；
- 运算的方式错误(即运算的对象彼此在类型上不相容、算法错误、初始化不正确、运算精度不够、表达式的符号不正确等)。

常见的比较和控制流错误有：

- 不同数据类型的比较；
- 不正确的逻辑运算符或优先次序；
- 因浮点运算精度问题而造成的两值比较不等；
- 关系表达式中不正确的变量和比较符；
- “差1错”。即不正确的多循环或少循环一次；
- 错误的或不可能的循环终止条件；
- 当遇到发散的迭代时不能终止的循环；
- 不适当地修改了循环变量等。

6.5.4 出错处理

比较完善的模块设计要求能预见出错的条件，并设置适当的出错处理，以便在程序出错时，能对出错程序重新做安排，保证其逻辑上的正确性。这种出错处理也是模块功能的一部分。

表明出错处理模块有错误或缺陷的情况有：

- 出错的描述难以理解；
- 出错的描述不足以对错误定位和确定出错的原因；
- 显示的错误与实际的错误不符；
- 对错误条件的处理不正确；
- 在对错误进行处理之前，错误条件已经引起系统的干预等。

6.5.5 边界条件

边界上出现的错误是常见的。

例如：

- 在 n 次循环的第 n 次，取最大值或最小值时容易发生错误；
- 特别要注意数据流，控制流中刚好等于、大于、小于确定的比较值时出现错误的

可能性。

6.6 单元测试用例设计思路

对于单元测试用例的设计,不能仅参照具体代码来进行,若这样可能会导致测试倾向于代码本身,即会被被测代码牵引测试用例的设计思路,而将被测试代码对应的设计文档抛置一边。测试用例的设计根据是软件的设计说明书,即详细设计文档。对于单元测试,测试用例用来证明一个独立的单元是否实现了单元设计说明书中的要求。一个完整的单元测试不仅仅要进行正向测试,即测试被测单元是否做了它应该做的事情(满足设计说明书要求),同时还应该做逆向测试,即被测单元有没有做并不希望它做的事情(即设计说明书以外的事情)。

下面是6个通用步骤,用来指导完成测试用例的设计^[39]。

6.6.1 为系统运行设计用例

单元测试方案中的第一个测试用例最有可能是用最简单的方法执行被测单元。当第一个测试用例可以正常进行,至少知道测试环境和被测试单元是可用的,如果不能执行,那么最好重新调试,也就是说还不具备开始测试的条件。

可使用的测试分析技术:

- 规范导出法;
- 等价类划分。

6.6.2 为正向测试设计用例

测试用例的设计者应该通读相关的设计说明,每一个测试用例,都是针对测试说明书中的一项或多项内容来设计的。当涉及到不止一个设计说明书时,最好把测试方案中的测试用例与主要的设计说明书中的描述顺序相对应。正向测试的用例就是验证设计说明书所对应的功能项或性能指标能否兑现。

可使用的测试分析技术:

- 规范导出法;
- 等价类划分;
- 状态转换测试。

6.6.3 为逆向测试设计用例

逆向测试的用例就是用来验证被测的软件单元有没有做它不应该做的事情。此步骤主要依靠错误猜测的方法进行测试用例的构造。

可使用的测试分析技术:

- 错误猜测法；
- 边界值分析；
- 状态转换测试。

6.6.4 为满足特殊需求设计用例

从系统的性能、安全性、保密性的角度来设计测试用例是十分必要的。尤其对于安全及保密要求较高的系统，在测试方案中，特别的标明用来进行安全及保密的测试用例是大有裨益的。

可使用的测试分析技术：规范导出法。

6.6.5 为代码覆盖设计用例

设计好的测试用例是可以保证较高的代码测试覆盖率的。在单元测试方案中，为了达到特定的测试覆盖目标，可能还需要补充一些测试用例。当为测试覆盖率而补充的测试用例设计好后，测试方案基本完成，就可以具体执行测试用例了。

可使用的测试分析技术：

- 分支测试；
- 条件测试；
- 数据定义使用测试；
- 状态转换测试。

6.6.6 为覆盖率指标完成设计用例

通过上面 5 个步骤设计出来的测试方案，在大多数情况下，对于一个单元，都能够提供较全面的测试。此时，我们可以开始测试了。

不幸的是，在被测试的代码中，可能包含有复杂的判断条件，循环以及分支语句，因此在执行测试用例的过程中，覆盖率的目标有可能无法达到。当这种情况发生时，有必要分析为什么会导覆盖率目标没有达到。一般的原因可能有以下几种：

- 不可能的路径或条件；
- 不可达的或冗余的代码；
- 不充分的测试用例。

可使用的测试分析技术：

- 分支测试；
- 条件测试；
- 数据定义使用测试；
- 状态转换测试。

大多数有效的测试用例都来自于分析，而不是仅仅为了达到测试覆盖率目标而草率设计的测试用例。

上面提到的测试分析技术可以参考本书第5章的内容。关于如何详细进行单元测试的实例可以参考笔者即将出版的书籍《软件测试实战》。

6.7 单元测试过程

有很多人认为单元测试是在代码完成之后才开始的，也有很多人认为单元测试就是执行测试用例。错！很多关于测试方面的研究都已经指明，测试应当越早开始越好，一个好的测试需要经过计划、设计、执行到报告总结多个过程^{[2][3][4][6][8][26][36][119][124][125][126]}。在此我们根据 ANSI/IEEE Std 1008-1987 的标准，把单元测试过程划分为3个阶段，8个活动，具体见图6-4。

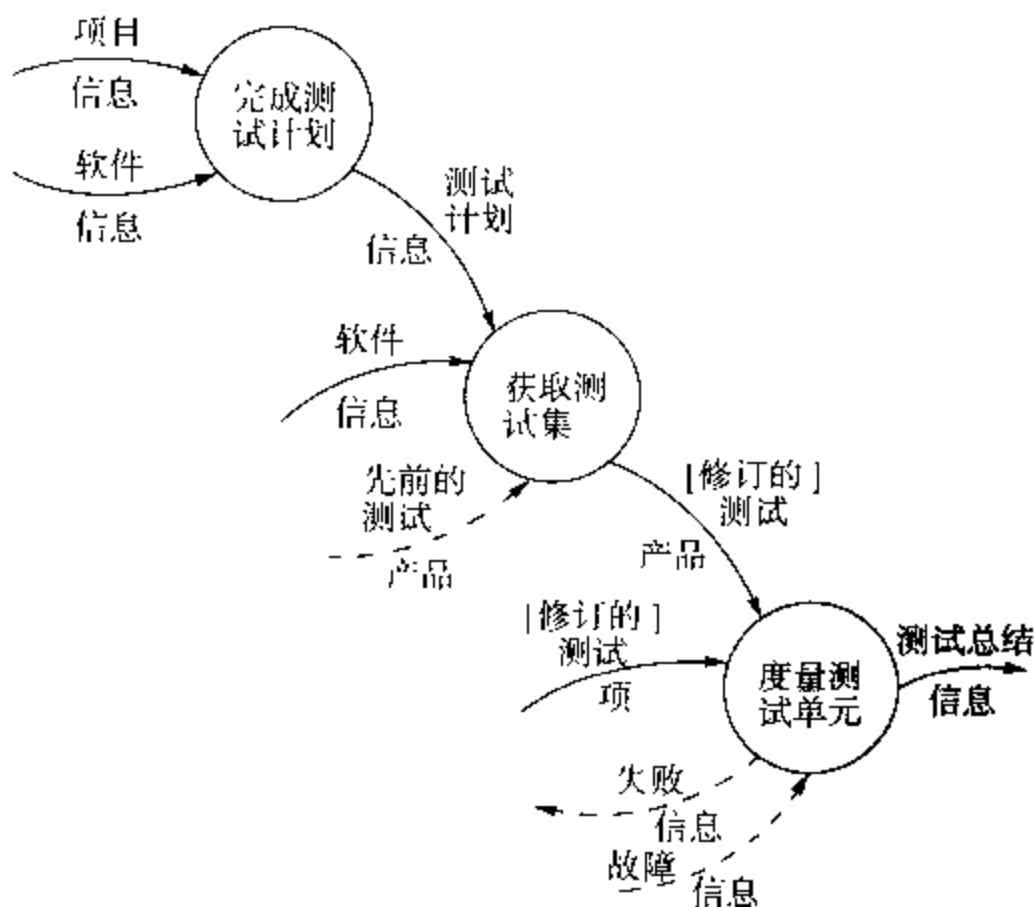


图 6-4 单元测试阶段主要数据流图

- 完成测试计划(计划阶段)
 - (1) 计划总的方法、资源和进度(活动1)。
 - (2) 确定被测特性(活动2)。
 - (3) 优化总的计划(活动3)。
- 获取测试集(设计阶段)
 - (1) 设计测试集(活动4)。
 - (2) 完成改进的计划和设计(活动5)。
- 度量测试单元(执行和总结阶段)
 - (1) 执行测试规程(活动6)。
 - (2) 检查结束条件(活动7)。
 - (3) 评价测试工作和单元(活动8)。

6.7.1 完成测试计划

1. 计划总的方法、资源和进度

一般来说,单元测试计划应当在整个测试计划制订期间产生,对应的开发阶段为设计阶段。

- 活动的输入

- (1) 项目计划。

- (2) 软件需求文档。

- 活动的任务

- (1) 指定一个单元测试的总的策略。

确定测试关注的风险区域。确定关于特性、测试设计或者测试实现的约束(如,必须使用的测试集)。

确定已有的输入、输出或状态数据源(例如,测试文档、产品文档、测试数据产生器)。确定用于数据验证的一般测试技术。确定用于输出记录、收集、裁减和验证的一般技术。描述用于应用软件的规定。

- (2) 指定单元测试完整性需求。

确定需要通过单元测试集覆盖的区域(例如:特性、过程、状态、函数、数据特性、指令)和每个区域需要被覆盖的程度。

当在软件开发过程中测试一个单元时,每个软件特性必须被一个单元测试用例覆盖或者有一个批准的例外。对于维护期间的软件,单元测试也是一样要坚持的。

在软件开发期间,当用一个过程语言测试一个已实现的单元时,每个可以到达或执行到的指令必须被一个单元测试用例覆盖到或者有一个批准的例外。对于维护期间的情况也是一样的。

- (3) 指定单元测试终止需求。

指定单元测试异常终止的需求。正常终止需求必须包含满足完整性测试需求。

确定每个可能引起单元测试过程异常终止的条件和任何应用的通知过程。

- (4) 确定单元测试资源需求。

估计测试集获取,最初的执行以及随后的测试活动回归需要的资源。考虑硬件、访问时间、通信或系统软件、测试工具、测试文档、表格和其他需要提供的东西。

- (5) 指定一个总的单元测试进度。

指定一个由资源和可获得的测试单元所约束的时间进度。

- 活动的输出

- (1) 总的单元测试计划信息。

- (2) 单元测试总的资源需求。

2. 确定被测特性

- 活动的输入

- (1) 单元需求文档。

(2) 软件构架设计文档——如果需要的话。

- 活动的任务

(1) 研究功能需求。

研究描述在单元需求文档中的每个功能描述。保证每个功能有一个惟一的标识。当必须的时候,需要对需求进行澄清。

(2) 确定额外的需求和相关的过程。

确认功能以外(如,性能、属性等)与软件相关且可以在单元级被有效测试的需求。确认任何只与被测单元相关的用法或操作过程。保证每个额外的需求和过程有一个惟一的标识。当必须的时候,需要对需求进行澄清。

(3) 确定单元的状态。

如果单元需求文档指定或者隐含多个状态(例如,停止、准备接受、处理等)软件,确认每个状态和每个有效状态转换。保证每个状态和状态转换有一个惟一的标识。当必须的时候,需要对需求进行澄清。

(4) 确定输入和输出数据特性。

确认被测单元的输入和输出数据结构。对每个结构,确认特性,例如到达的速率、格式、值范围和域值之间的关系。对每个特性,制订它的有效范围。保证每个特性有一个惟一的标识。当必须的时候,需要对需求进行澄清。

(5) 选择测试中被包含的元素。

选择被测特性。选择相关的过程、相关的状态、相关的**状态变换**和相关的**数据特性**,它们被测试所包含。必须选择有效和无效的数据。当完整的**测试不现实**时,单元期望使用的信息被用于确定这个选择。确认没有选择元素的风险。把**被选择的特性**、过程、状态、状态转换和数据特性记录到单元测试设计规格的“被测特性”一节中。

- 活动的输出

(1) 测试中被包含的元素列表。

(2) 单元需要澄清的需求。

3. 优化总的计划

- 活动的输入

(1) 测试包含的元素列表。

(2) 总的单元测试计划信息。

- 活动的任务

(1) 优化方法。

确认被考虑使用的已有测试用例和测试规程。确认任何**被用于数据验证**的特殊技术。确认任何被用于输出记录、收集、简化和验证的特殊技术。该**优化的方法**记录在单元测试设计规格的“方法优化”一节。

(2) 指定特殊资源需求。

确认任何测试单元需要的特殊资源(例如,直接和单元接口的软件)。为确认的资源做准备。该特殊资源需求记录在单元测试设计规格的“方法优化”一节。

(3) 指定一个详细的进度。

制订一个用于单元测试的进度，该进度基于支持软件、特殊资源和可用的单元及集成进度。该进度记录在单元测试设计规格的“方法优化”一节。

- 活动的输出

- (1) 指定的单元测试计划信息。

- (2) 单元测试特殊需求。

6.7.2 获取测试集

1. 设计测试集

- 活动的输入

- (1) 单元需求文档。

- (2) 测试包含的元素列表。

- (3) 单元测试计划信息。

- (4) 单元设计文档。

- (5) 先前测试的测试规格——如果有的话。

- 活动的任务

- (1) 设计测试集结构。

基于被测特性和由被选择的相关元素指定或隐含的条件(例如，过程、状态转换、数据特性)，设计一个分层的测试对象集以便每个最低级别的对象可以直接被一些测试用例测试。选择适当已有的测试用例，把测试用例组标号和最低级别对象关联起来。在单元设计规格的“测试确认”一节记录对象的层次和关联的测试用例标号。

- (2) 获得需要的清晰的测试规程。

一个单元需求文档、测试计划信息和测试用例规格的组合可能隐含的指定了单元测试的过程，并且因此最小化用于清晰规格的需要。选择已有的可以被修改或不需要被修改的测试规程。在单元测试设计文档的“补充”一节指定任何需要的额外过程。

- (3) 获取测试用例规格。

指定新的测试用例。已有的规格可以被引用。在单元测试设计文档的“补充”一节记录直接的规格或被引用的规格。

- (4) 根据需要，基于设计信息扩大测试用例规格集。

根据单元设计的信息，更新需要的测试集结构。考虑已选择的算法和内部数据结构的特点。确认控制流和必须被记录的对内部数据的变更。预计不同于可能产生的特殊记录，例如，来自于在复杂算法中的跟踪控制流的需要或来自于跟踪内部数据结构变更的需要。必要时，要求加强单元测试设计以增加单元的可测试性。基于单元设计的信息，指定任何新的确定的测试用例和完成任何局部的测试用例规格。

- (5) 完成测试设计规格。

完成用于单元的测试设计规格。

- 活动的输出

- (1) 单元测试设计规格。

- (2) 独立的测试规程规格——如果有的话。

(3) 独立的测试用例规格——如果有的话。

(4) 单元设计加强需要——如果有的话。

2. 完成改进的计划 and 设计

- 活动的输入

(1) 单元测试计划信息。

(2) 测试用例规格。

(3) 软件数据结构描述。

(4) 测试支持资源。

(5) 测试项。

(6) 来自于先前测试活动的测试数据——如果有的话。

(7) 来自于先前测试活动的测试工具——如果有的话。

- 活动的任务

(1) 获取和验证测试数据。

获取一个已有测试数据，经过修改或没有修改过的拷贝，产生任何新的数据，包括额外的保证数据一致性的必须的数据。根据软件数据结构规格验证所有数据。当测试用例和数据集之间联系不是很明显的时候，开发一个表格用于记录这个相关性并把它包含到单元测试设计文档中。

(2) 获取特殊的资源。

获取前面已经确定的支持资源。

(3) 获取测试项。

收集测试项，包括可获得的手册、操作系统过程、控制数据和计算程序。获取在测试计划期间确认的软件以及这些软件与被测单元有直接的接口。当用一个过程语言测试一个实现了的单元时，保证执行跟踪的信息可以用于评价基于代码的完整性需求是否可以被满足。把每个项确认的结果记录到单元测试总结报告的“总结”一节。

- 活动的输出

(1) 验证了的测试数据。

(2) 测试支持资源。

(3) 测试项的配置。

(4) 原始总结信息。

6.7.3 度量测试单元

1. 执行测试规程

- 活动的输入

(1) 验证了的测试数据。

(2) 测试支持资源。

(3) 测试项的配置。

(4) 测试用例规格。

(5) 测试规格规格——如果已经产生。

(6) 失败分析结果——如果已经产生。

- 活动的任务

(1) 运行测试。

设置测试环境，运行测试集，记录所有测试意外到单元测试总结报告的“结果总结”一节中。

(2) 确定结果。

对每个测试用例，根据在用例描述中期望的结果规格确定单元通过还是失败。在单元测试总结报告的“结果总结”一节中记录通过或失败结果。在报告的“活动总结”一节记录消耗了的资源数据。当使用一个过程语言测试一个已实现的单元时，收集执行跟踪总结信息并把该信息附加到报告中。对于每个失败，需要有失败分析并把故障信息记录到测试总结报告的“结果总结”一节中。然后选择可用的情况并执行相关的活动，这些情况和活动如下所述。

情况 1 在测试规格或测试数据中的一个缺陷。修正缺陷，在测试总结报告的“活动总结”一节记录纠正的缺陷并且重新运行失败的测试。

情况 2 在测试规程执行中的缺陷。重新运行被执行的不正确的规程。

情况 3 在测试环境中的一个故障。要么纠正环境，在总结报告中记录纠正的故障，并且重新运行失败的测试；要么准备异常终止并文档化不纠正环境问题的原因。

情况 4 在单元实现中的一个缺陷。要么纠正单元，在总结报告中记录纠正的缺陷，并且重新运行失败的测试；要么准备异常终止并文档化不纠正单元缺陷的原因。

情况 5 在单元设计中的一个缺陷。要么纠正设计和单元，更改相关的测试规格和数据，在总结报告中记录纠正的缺陷，并且重新运行失败的测试；要么准备异常终止并文档化不纠正设计缺陷的原因。

- 活动的输出

(1) 在总结报告中的执行信息日志(包括测试结果、测试异常描述、失败分析结果、缺陷纠正活动、未纠正缺陷的原因、资源消耗数据和对过程语言实现的跟踪总结信息)。

(2) 修正过的测试规格。

(3) 修正过的测试数据。

2. 检查结束条件

- 活动的输入

(1) 完整性和终止需求。

(2) 执行信息。

(3) 测试规格——如果需要。

(4) 软件数据结构描述——如果需要。

- 活动的任务

(1) 检查测试过程的正常终止。

确定基于完整性需求的用于额外测试的需要，或者确定由失败历史而引起的需要关注的需求。对于由过程语言实现的，分析执行跟踪总结信息。

如果不需要额外测试,那么在测试总结报告的“活动总结”一节记录正常终止,并且对测试工作和单元进行评价。

(2) 检查测试过程的异常终止。

如果异常终止条件被满足,那么保证引起终止的特定情况被文档化到测试总结报告的“活动总结”一节,连同没有完成的测试和任何未纠正的缺陷。然后,对测试工作和单元进行评价。

(3) 补充测试集。

当需要额外的测试并且异常终止条件没有被满足时,根据下面步骤补充测试集:

① 按照6.7.2节中第1点的活动的第一个任务方法更新测试集结构,按照6.7.2节中第1点的活动的第三个任务方法获取额外测试用例规格

② 按照6.7.2节中第1点的活动的第二个任务方法修改测试规程规格

③ 按照6.7.2节中第2点的活动的第一个任务方法获取额外测试数据

④ 在测试总结报告的“活动总结”一节记录增加的内容

⑤ 执行这些额外的测试

- 活动的输出

(1) 在测试总结报告中记录的检查信息,包括终止条件和任何测试用例附加活动。

(2) 额外的或修正过的测试规格——如果有的话。

(3) 额外的测试数据——如果有的话。

3. 评价测试工作和单元

- 活动的输入

(1) 单元测试设计规格。

(2) 执行信息。

(3) 检查信息。

(4) 独立的测试用例规格——如果产生的话。

- 活动的任务

(1) 描述测试状态。

在测试总结报告的“变化”一节记录与测试计划和测试规格的不同地方。对每个不同指出原因。对于异常终止,在测试总结报告的“完整性评价”一节中确认测试覆盖不充分的区域并记录原因。在测试总结报告的“结果总结”一节中确认未解决的测试异常和原因。

(2) 描述单元的状态。

在测试总结报告的“变化”一节记录通过测试揭露的单元和它的规格文档不同的地方。针对需求,根据测试结果和被检测到的缺陷信息评价单元的设计和实现。在测试总结报告的“评价”一节记录评价信息。

(3) 完成测试总结报告。

根据报告的标准完成单元的测试总结报告。

(4) 保证测试产品的保存。

保证测试产品被收集、组织和保存,以便今后被重用。产品包括测试设计规格、独立的测试用例规格、独立的测试规程规格、测试数据、测试数据产生过程、测试驱动和桩,

以及测试总结报告。

- 活动的输出

(1) 完整的测试总结报告。

(2) 完整的被保存的测试产品。

6.8 单元测试工具介绍

目前市场上的测试工具有很多, 哪些工具适合于单元测试呢? 常见的可用于单元测试的工具包括:

- 代码静态分析工具

Logiscope, McCabe QA, CodeTest 等

- 代码检查工具

PC - LINT, CodeChk, Logiscope 等

- 测试脚本工具

TCL, Python, Perl 等

- 覆盖率检测工具

Logiscope, PureCoverage, TrueCoverage, McCabe Test, CodeTest 等

- 内存检测工具

Purify, BoundsCheck, CodeTest 等

- 专为单元测试设计的工具

RTRT, Cantata, AdaTest 等

关于软件测试工具的详细介绍请参考笔者即将出版的《软件测试自动化》一书。

6.9 单元测试应坚持的原则

对于测试来说, 应当尽早和不断地进行软件测试。对于单元测试来说需要遵循一定的单元测试规范, 下面列出了一些基本性原则, 但是这些并不是足够的, 读者可以结合自己的经验补充这些规定:

- 对全新的代码或修改过的代码进行单元测试;
- 被测试的对象为实现一组相关功能的代码(一个或一组函数);
- 单元测试根据单元测试计划和方案进行, 排除测试的随意性;
- 项目管理者保证测试用例经过审核;
- 当测试用例的测试结果与预期结果不一致时, 单元测试的执行人员需如实记录实际的测试结果;
- 当测试计划中的结束标准达到时, 单元测试结束;
- 对被测试单元需达到的一定的代码覆盖率要求;
- 当程序进行了修改, 则测试执行人员执行回归测试以保证对发现错误的修改没有

引入新的错误。

6.10 我们的问题

笔者接触过一些国内的软件公司，也与许多软件测试同行讨论过单元测试技术方面的问题。从现在的情况来看，有许多公司还没有意识到单元测试的重要性；因此，也没有相应的单元测试活动。对于已经有单元测试活动的公司来说，或多或少还存在着下面一些问题：

(1) 单元测试过程等到编码结束后才开始。

在本章6.7节我们已经讨论过这个问题。

(2) 单元测试用例是根据代码，而不是根据需求规格和详细设计说明书设计的。

建议设计人员和编码人员分离，或者进行交叉设计。

(3) 满足于单纯的100%语句覆盖，为测试而测试，并未针对出错概率高的代码设计专门测试用例。

不能把100%的语句覆盖当作一个紧箍咒，应当充分结合等价类划分、边界值选取等测试用例设计方法来进行用例的设计。如果使用的方法是全面的，那么即使最后的测试结果没有达到100%语句覆盖也应当是允许的。如果一味地追求100%的语句覆盖，而不管用例的设计方法，那么就已经走入歧路了。

(4) 没有严格地选择被测单元

单元的选择也会影响单元测试的效率，单元太大，那么内部逻辑结构就会越复杂，覆盖率的指标就很难达到，等价类的分析也会变的很复杂，这样会使得测试人员很疲惫，最后效果也不大。单元选择太小，太简单，为这类代码写方案、写用例、写驱动等就有些得不偿失，一般这类代码采用走读方式比较适合。那么单元到底该怎么选取呢？没有什么绝对的原则，一般以函数为单位，根据本产品特点和代码复杂度情况选取合适单元进行测试。

(5) 开发人员对单元测试技术方法的了解和掌握普遍匮乏。

(6) 在进行驱动模块、桩模块和测试用例设计的时候未充分考虑测试的可回归性，自动化程度不高。

提高单元测试的自动化程度可以提高单元测试的可回归性，人工干预越多，回归性越差。俗话说：一鼓作气，再而衰，三而竭。况且单元测试可能会进行很多次，因此一定要提高单元测试的自动化程度。有很多方法可以选用，也可以借助一定的工具。

(7) 没有有效使用一些好的测试工具。

充分使用各类测试工具能够有效地提高测试的质量。有人建议能否提供一个统一的工具？很难。单元测试情况千变万化，复杂程度各不相同，统一工具谈何容易。事实上统一程度越高，其实现的功能就越弱。一般在测试中我们比较关心代码测试工具、代码静态分析工具、代码动态覆盖率测试工具、回归测试工具、内存泄露检测工具、脚本解释工具等。充分利用各工具优点，结合使用可以最大限度提高测试的质量。

(8) 牺牲单元测试效率来换取项目进度。

尤其是在受项目进度的影响下，很多开发人员不得不牺牲单元测试的效率，缩短单元测试的时间来保证项目进度不受影响。这种做法是利还是弊？不可否认，我们的产品是以利益为主要目的，抢占市场就意味着成功。一个没有市场的优秀产品是不会有效益的。但没有质量的产品是否能长期占有市场呢？这里，我们不必来争论谁对谁错，我们的目的是要在有限的时间内让我们的工作做得最好。如果能够在实际上证明延长一定的单元测试时间并提高单元测试的效率可以缩短集成测试和系统测试的时间，提高整个系统的质量，那么可以相信许多产品经理和开发经理是会接受这种选择的。

(9) 如何评价单元测试的效果？

有人在用例设计过程或内部逻辑分析过程中找出了问题，并修改了问题，但是却没有记录，这样最后执行用例时发现的问题少了。这是不是单元测试没有效果了呢？不是！在单元测试过程中，利用单元测试方法发现的问题都应当记录到单元测试结果中，这是为了衡量单元测试效果，同时也是为了总结经验，进行改进。

6.11 本章小结

单元测试属于白盒测试范畴，是所有动态测试中粒度最小的测试，也是验证活动中的基础活动。一般的单元执行策略有3种：

- 由顶向下进行单元测试，该方法可以省去驱动模块的设计；
- 由底向上进行单元测试，该方法可以省去桩模块的设计；
- 独立的单元测试，及不与任何模块发生关系，所有需要用到的其他单元都做桩模块，驱动模块也自己设计。

方法1和2综合了集成的概念，随着单元测试的进行，可以看到系统一个初步集成的概貌。但是覆盖率会越来越难以保证，并且在每个单元测试之前必须保证相关单元的正确性。方法3比较独立，覆盖率容易保证，并且可以并行进行，但工作量最大。一般采用混合方法比较好。在对一个单元进行分析时，可以从5个角度进行考虑：模块接口、局部数据结构、独立路径、出错处理和边界条件。根据这些分析，可以应用第5章中提到的测试分析方法产生测试数据。一般常用的用例设计方法包括：规范导出法、等价类划分、边界值分析、数据流分析、错误猜测，根据覆盖率进行设计、因果图和判定表等。

单元测试是有一个过程的，并且该过程绝对不是在代码完成之后开始的。单元测试最早开始的时间可以追溯到设计阶段。一般一个单元测试过程可以划分为计划阶段、设计阶段和执行总结阶段。在有些文档中把执行阶段和总结阶段分开了，也有一些文档把设计阶段划分为用例设计阶段和脚本实现阶段。单元测试计划阶段和设计阶段是一个很重要的阶段，国内很多软件公司都不是很重视这个阶段。这两个阶段直接决定了单元测试的质量。从经验来看，单元测试执行和总结阶段基本和编码阶段时间差不多或略有减少，但是单元测试计划阶段和设计阶段将占整个单元测试过程的60%~70%左右的时间。

第7章 集成测试

单元测试 → 集成测试 → 系统测试，是一个逐步扩大的测试，由最小的单元级到最后的全系统级别。如果说软件开发过程是经由一个从需求 → 高层设计 → 底层设计 → 编码的一个逐步细化过程的话，那么从单元测试到系统测试的过程是对系统的一个逆向求证过程。集成测试处在从单元到系统的过渡阶段，对应于开发的概要设计。本章我们将学习集成测试方面的知识，通过本章的学习，你将了解：

1. 什么是集成测试？
2. 常见的集成测试策略有哪些？
3. 什么是自底向上集成，自顶向下集成和三明治集成？
4. 基于集成的特点是什么？
5. 对于面向对象软件可以使用哪些集成策略？
6. 集成测试分析可以从哪几个角度进行？
7. 常见的集成测试故障有哪些？
8. 如何设计集成测试用例？
9. 一个集成测试的过程包含哪些阶段，哪些活动，有哪些输入和输出？
10. 进行集成测试需要遵循哪些原则？

一般来说，软件测试可以分为白盒测试和黑盒测试两大阵营（参考第2章内容），而集成测试处于这两大阵营的边缘，可以说是结合了这两者的特点。尽管很多资料都将其划分到黑盒测试范畴，但是现在越来越多的学者将其定义为灰盒测试（Gray Box Testing）

7.1 什么是集成测试

集成测试（Integration Testing），也叫组装测试、联合测试、子系统测试或部件测试。集成测试是在单元测试的基础上，将所有模块按照概要设计要求（如根据结构图）组装成为子系统或系统，进行集成测试。也就是说，在集成测试之前，单元测试已经完成，并且集成测试所使用的对象应当是已经经过单元测试保证了的单元，如果不经单元测试，那么集成测试的效果将受到影响，并且会付出更大的代价，就如第6章所说的，单元测试和集成测试所关注的范围是不同的，因此，它们在发现问题的集合上包含有不相交的区域，不能使用单元测试来替代集成测试，反之也是一样。

7.1.1 集成测试与系统测试的区别

系统测试所测试的对象是整个系统以及与系统交互的硬件和软件平台。系统测试更多

程度上是站在用户的角度上对系统做功能性的验证,同时还对系统进行一些非功能性的验证,包括性能测试、压力测试、容量测试、安全性测试、恢复性测试等。系统测试的依据来自于用户的需求规格说明书和行业的已成文的或事实上的标准。

集成测试所测试的对象是模块间的接口,其目的是要找出在模块接口上面,包括整体体系结构上的问题。其测试的依据来自于系统的高层设计(构架设计)。

7.1.2 集成测试关注的重点

实践表明,一些模块虽然能够单独地工作,但并不能保证连接起来也能正常工作。程序在某些局部反映不出来的问题,在全局上很可能暴露出来,影响功能的实现。因此集成测试应当考虑以下问题^[2]:

- 在把各个模块连接起来时,穿越模块接口的数据是否会丢失。
- 各个子功能组合起来,能否达到预期要求的父功能。
- 一个模块的功能是否会对另一个模块的功能产生不利的影响。
- 全局数据结构是否有问题,会不会被异常修改。
- 单个模块的误差积累起来,是否会放大,从而达到不可接受的程度。

因此,单元测试后,有必要进行集成测试,发现并排除在模块连接中可能发生的上述问题,最终构成要求的软件子系统或系统。

7.1.3 集成测试和开发的关系

在第1章我们提到了V模型(参考图1-3),从图中可以看到集成测试是与软件开发的概要设计阶段相对应的,软件概要设计中关于整个系统的体系结构是集成测试的输入基础。体系结构是把一个大的系统组织成为可以管理的和可实现的组件或子系统的结构(见图7-1,图7-2,图7-3)。它服务于很多目标,包括风险管理、进度和验证。Booch认为集成是面向对象开发中最关键的活动^[147]。其实即使在结构化设计中,集成也是同样重要的。

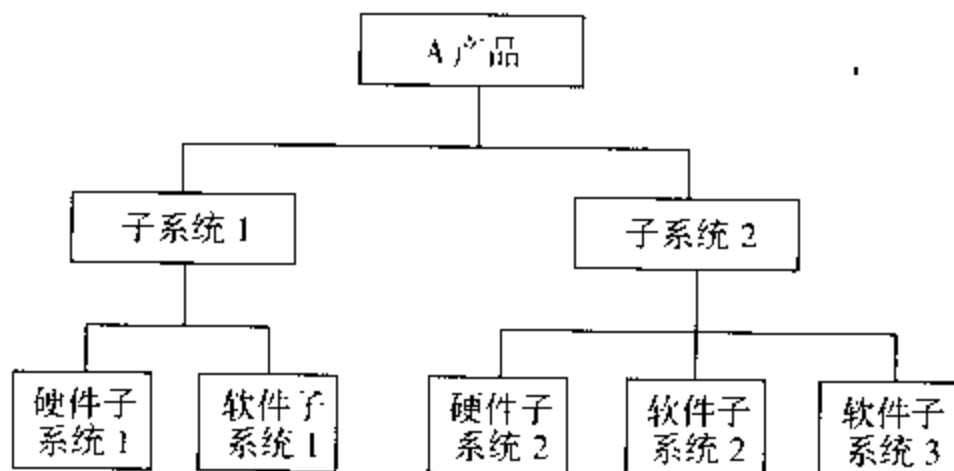


图 7-1 系统结构图



图 7-2 软件结构图

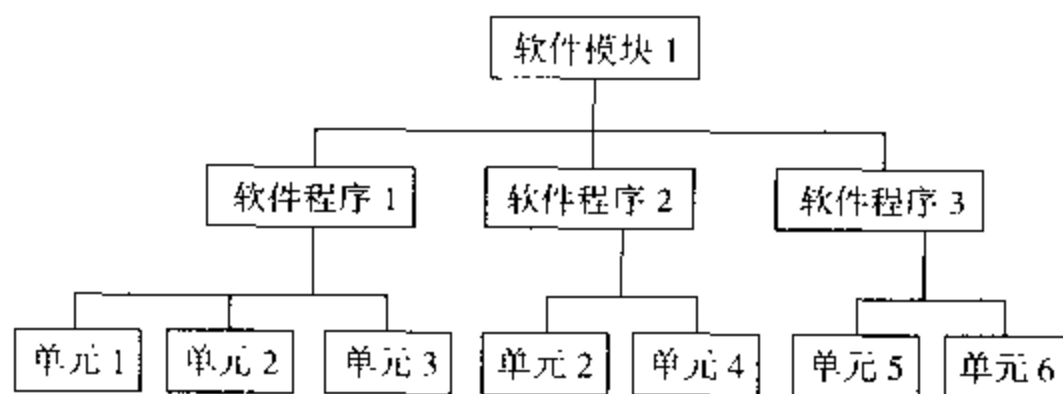


图 7-3 软件模块结构图

集成测试与构架设计之间具有相互的依赖性。如果构架设计不明确,集成测试设计将无法很好地完成。同样集成测试可以用来发现构架设计中的错误、遗漏和二义性问题,包括前期的验证活动和后期的确认测试。

7.1.4 集成测试的层次

一个产品的开发过程包括了一个分层的设计和逐步细化的过程,从最初的产品到最小的单元。该层次大致可以通过图 7-1、图 7-2、图 7-3 表示出来。

一般单元级测试针对该体系结构图最底层的叶子节点,而系统测试对应的是产品级的。其中所有各层测试都需要通过集成测试来完成,由于集成的力度不同,一般可以把集成测试划分成 3 个级别:

- 模块内集成测试;
- 子系统内集成测试;
- 子系统间集成测试。

不同级别的测试所使用的集成策略也会有所不同,具体集成测试策略的内容请参考 7.2 节内容。

7.2 集成测试策略

集成测试策略就是在测试对象分析的基础上,描述软件模块集成(组装)的方式、方法。集成测试的基本策略比较多,分类比较杂,Mayer 分析比较了自底向上集成(Bottom-Up Integration)、自顶向下集成(Top-Down Integration)、大爆炸集成(Big Bang Integration)和三明治集成(Sandwich Integration)^{[2][126]}。Beizer 提出了基于集成(Backbone Integration)方法^[26]。Jacobson 提出了关于面向对象系统中集成的策略^[128]。Robert V. Binder 对这些方

法进行了综合性的介绍，并总结成了相应的模式^[127]。下面我们分别来介绍。

7.2.1 大爆炸集成

1. 目的

在最短的时间内把系统组装出来，并且通过最少的测试来验证整个系统。

2. 介绍

大爆炸集成(Big Bang Integration)是属于非增值式集成(Non-Incremental Integration)的一种方法，也叫一次性组装或整体拼装。该集成把所有系统组件一次性集合到被测系统中，不考虑组件之间的相互依赖性或者可能存在的风险。应用一个系统范围内的测试包来证明系统最低限度的可操作性。

3. 策略

在这种集成方式中，首先对每个模块分别进行单元测试，然后再把所有单元组装在一起进行测试，最终得到要求的软件系统。如图 7-4(a)有一系统结构，其单元测试和组装顺序如图 7-4(b)所示。

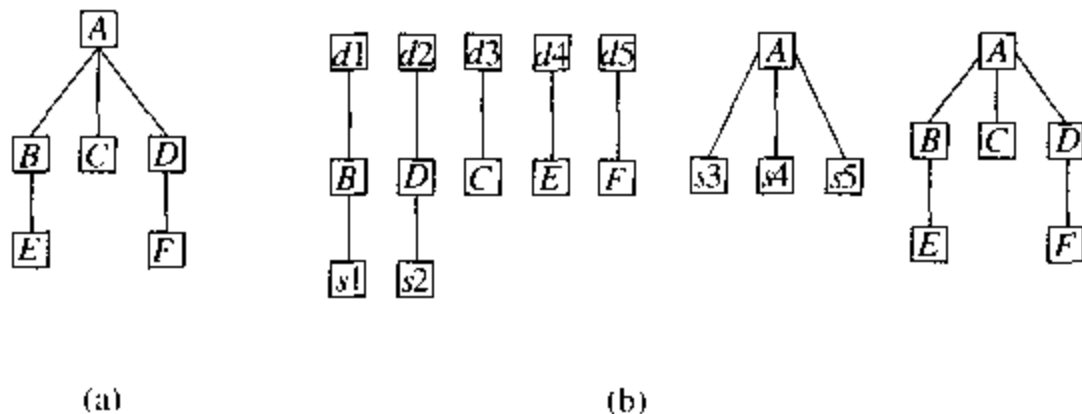


图 7-4 一次性组装示意图

在图 7-4 中，模块 $d1$ 、 $d2$ 、 $d3$ 、 $d4$ 、 $d5$ 是对各个模块做单元测试时建立的驱动模块， $s1$ 、 $s2$ 、 $s3$ 、 $s4$ 、 $s5$ 是为单元测试而建立的桩模块。

4. 优点

- 在有利的情况下，大爆炸集成可以迅速完成集成测试，并且只要极少数的驱动和桩模块设计(如果需要的话)；
- 它需要的测试用例也是最少的；
- 该方法比较简单；
- 多个测试人员可以并行工作，对人力、物力资源利用率较高。

5. 缺点

- 这种一次性组装方式试图在辅助模块的协助下，在模块单元测试的基础上，将所

测模块连接起来进行测试。但是由于程序中不可避免地存在模块间接口、全局数据结构等方面的问题,所以一次试运行成功的可能性并不是很大;

- 在发现错误时,其问题定位和修改都比较困难;
- 即使被测系统能够被一次性集成,但还是会有许多接口错误很容易躲过测试而进入到系统范围测试内。

我们一般不推荐单独使用这种集成方法。

6. 适用范围

- 一个维护型项目(或功能增强型项目),其以前的产品已经很稳定,并且新增的项目只有少数几个组件被增加或修改;
- 被测系统比较小,并且它的每个组件都经过了充分的单元测试;
- 产品使用了严格的净室软件工程过程,并且每个开发阶段的质量和单元测试质量都相当高。

7.2.2 自顶向下的集成

1. 目的

从顶层控制开始,采用同设计顺序一样的思路对被测系统进行测试,以验证系统的接口稳定性。

2. 介绍

自顶向下的集成(Top-Down Integration)采用了和设计一样的顺序对系统进行测试,它在第一时间对系统的控制接口进行了验证。假定你正遵循一个迭代式或增量式的方法在开发一个系统,该系统控制结构模型与树一样,其中顶层的组件具有控制的责任,采用自顶向下的集成测试首先集中于顶层的组件,然后逐步测试处于底层的组件。自顶向下的集成方式可以采用深度优先策略和广度优先策略。

3. 策略

在此我们使用 7-4(a) 这个模型来描述该方法的具体策略:

- 以主模块为所测模块兼驱动模块,所有直属于主模块的下属模块全部用桩模块替换,对主模块进行测试;
- 采用深度优先(Depth-First)(如图 7-5 所示)或广度优先(Breadth-First)(如图 7-6 所示)的策略,用实际模块替换相应桩模块,再用桩代替它们的直接下属模块,与已测试的模块或子系统组装成新的子系统;
- 进行回归测试(即重新执行以前做过的全部测试或部分测试),排除组装过程中引起的错误的可能;
- 判断是否所有的模块都已组装到系统中。如果是,结束测试,否则转到步骤 2 去执行。

其中, s1、s2、s3、s4、s5 代表桩模块,图形中的集成顺序为自左到右,由上到下。

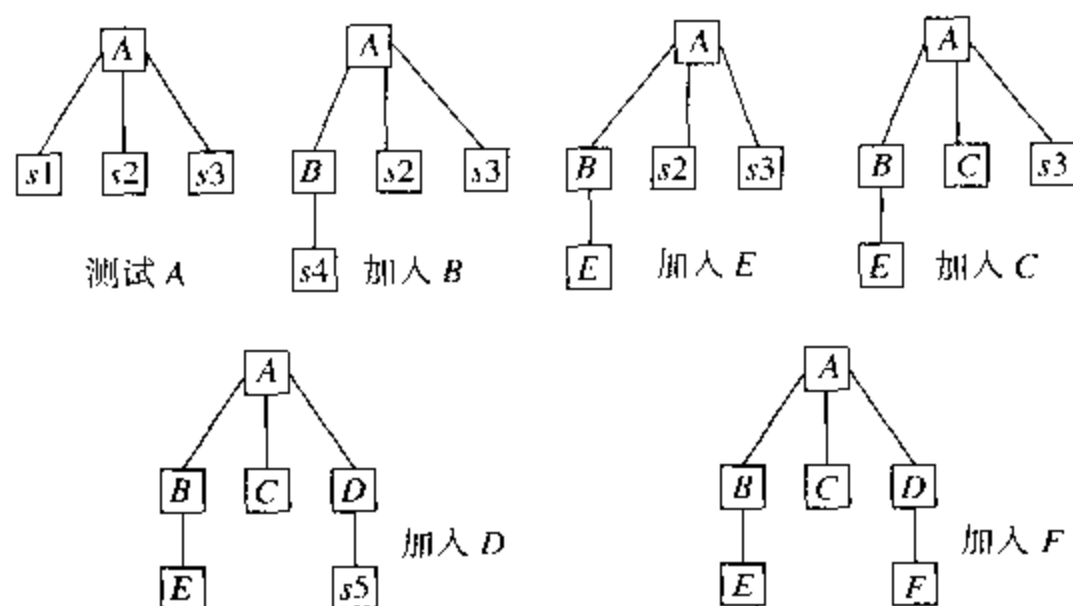


图 7-5 深度优先组装方式

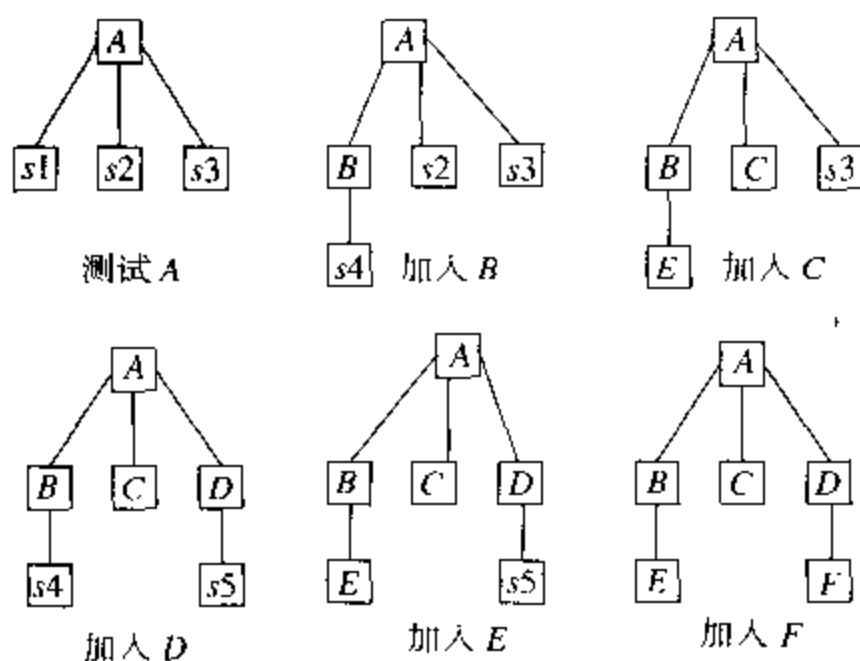


图 7-6 广度优先组装方式

4. 优点

- 自顶向下的增殖方式在测试过程中较早地验证了主要的控制和判断点。在一个功能划分合理的程序模块结构中，判断常常出现在较高的层次中，因而较早就能遇到。如果主要控制有问题，尽早发现它能够减少以后的返工，所以这是十分必要的；
- 如果选用按深度方向组装的方式，可以首先实现和验证一个完整的软件功能，可先对逻辑输入的分支进行组装和测试，检查和克服潜藏的错误和缺陷，验证其功能的正确性，就为其后对主要加工分支的组装和测试提供了保证；
- 功能可行性较早得到证实，还能够给开发者和用户带来成功的信心；
- 最多只需要一个驱动模块（顶层组件的驱动器）。减少了驱动器开发的费用。特定组件的驱动器一般使用难以编码的测试用例并且与组件的接口耦合性很大，这种设置限制了驱动器和测试包的复用。而采用自顶向下的策略，最多只需要维护一个顶层模块的驱动器，尽管也会遇到不可复用的问题，但维护工作量将小很多；

- 由于和设计顺序的一致性，因此可以和设计并行进行，如果目标环境可能存在改变，该方法可以比较灵活地适应；
- 支持故障隔离，例如，假设 *A* 模块执行正确，但是加入 *B* 模块后，测试执行失败，那么可以确定，要么 *B* 模块有问题，要么 *A* 和 *B* 的接口有错误。

5. 缺点

- 桩的开发和维护是本策略的最大成本。因为桩在每个测试中都必须被提供，并且随着测试配置使用的桩的数目增加，维护桩的成本将急剧上升；
- 底层组件中的一个无法预计的需求可能会导致许多顶层组件的修改，这破坏了部分先前构造的测试包；
- 底层组件行为的验证被推迟了。同时为了能够有效地进行测试，需要控制模块具有比较高的可测试性；
- 随着底层模块的不断增加，整个系统越来越复杂，导致底层模块的测试不充分，尤其是那些被重用的模块。

6. 适用范围

该方法适合于大部分采用结构化编程方法的软件产品，且产品的结构相对比较简单，一般对于大型复杂的项目往往会采用多种集成测试方法的综合使用策略。对于具有如下属性的产品，可以优先考虑本集成测试策略。

- 产品控制结构比较清晰和稳定；
- 产品的高层接口变化比较小；
- 产品的底层接口未定义或经常可能被修改；
- 产品的控制模块具有较大的技术风险，需要尽早被验证；
- 希望尽早能够看到产品的系统功能行为；
- 在极端编程(Extreme Programming)中使用探索式开发风格时，其集成策略可以采用自顶向下策略。

7.2.3 自底向上的集成

1. 目的

从具有最小依赖性的底层组件开始，按照依赖关系树的结构，逐层向上集成，以检测整个系统的稳定性。

2. 介绍

自底向上的集成(Bottom-Up Integration)方式是从程序模块结构的最底层的模块开始组装和测试。因为模块是自底向上进行组装，对于一个给定层次的模块，它的子模块(包括子模块的所有下属模块)已经组装并测试完成，所以不再需要桩模块。在模块的测试过程中想从子模块得到信息可以通过直接运行子模块得到。

3. 策略

自底向上集成的步骤如下：

- 起始于模块依赖关系树的底层叶子模块，也可以把两个或多个叶子模块合并到一起进行测试，或者把只有一个子节点的父模块与其子模块结合在一起进行测试；
- 使用驱动模块对步骤1选定的模块(或模块组)进行测试；
- 用实际模块代替驱动模块，与它已测试的直属子模块组装成为一个更大的模块组进行测试；
- 重复上面的行为直到系统的最顶层模块被加入到已测系统中。

以图 7-4(a) 这个模型为例，该集成策略可以使用图 7-7 表示。

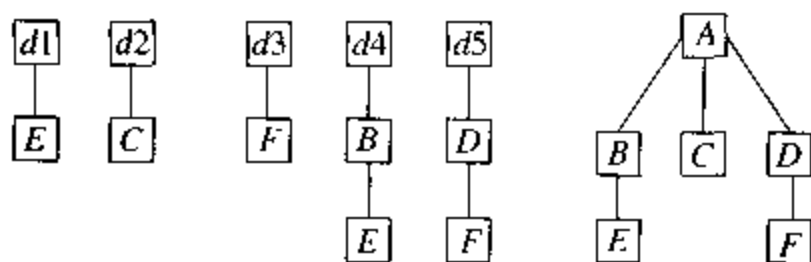


图 7-7 自底向上集成示意图

其中， $d1$ 、 $d2$ 、 $d3$ 、 $d4$ 、 $d5$ 代表驱动模块，图形中的集成顺序为由左到右。

4. 优点

- 允许对底层模块行为的早期验证。可以在任何一个叶子节点已经就绪的情况下进行集成测试；
- 在工作的最初可能会并行进行集成，在这一点上比使用自顶向下的策略效率高；
- 由于驱动模块是额外编写的，而不是实际模块，因此对实际被测模块的可测试性要求比自顶向下的集成策略要小得多；
- 减少了桩模块的工作量，毕竟在集成测试中，桩模块的工作量远比驱动模块的工作量要大得多。但是为了模拟一些中断或异常，可能还是需要设计一定的桩模块；
- 该方法也支持故障隔离。

5. 缺点

- 驱动模块的开发工作量也是很庞大的(可以通过提供对已测试组件的复用技术来降低这个成本)^{[2][126][130]}；
- 对高层的验证被推迟到了最后，设计上的错误不能被及时发现，尤其对于那些控制结构在整个体系中非常关键的产品；
- 随着集成到了顶层，整个系统将变得越来越复杂，并且对于底层的一些异常将很难覆盖，而使用桩将简单得多。

6. 适用范围

该方法适合于大部分采用结构化编程方法的软件产品，且产品的结构相对比较简单，

一般对于大型复杂的项目往往会采用多种集成测试方法的综合使用策略。对于具有如下属性的产品，可以优先考虑本集成测试策略。

- 采用契约式设计 (Design by Contract) 的产品^{[129][131]}；
- 底层接口比较稳定的产品；
- 高层接口变化比较频繁的产品；
- 底层模块较早被完成的产品。

注：契约式设计 (Dbc) 是一种使用注释来合并规格信息到代码本身的形式化方法。基本上，代码规格被使用一种用于描述代码隐含契约的形式化语言明确的表达。这些契约指出了类似下面的需求：

- 一个方法被调用前，客户端必须满足的条件
- 方法在被执行后必须被满足的条件
- 在方法满足上面条件的特定执行点上加入断言

7.2.4 三明治集成

1. 目的

综合自顶向下的集成测试策略和自底向上底集成测试策略优点。

2. 介绍

三明治集成 (Sandwich Integration) 有时也被称为混合式集成。由于自顶向下集成策略和自底向上底集成策略都有各自的缺点，因此自然而然地想到集中这两者优点的混合测试策略。三明治集成就是这样一种方法，它把系统划分成三层，中间一层为目标层。测试的时候，对目标层上面的一层使用由顶向下的集成策略，对目标层下面的一层使用自底向上的集成策略，最后测试在目标层会合。

3. 策略

使用 7-4(a) 这个模型，其中目标层为 B, C, D ，目标层上面一层是 A ，目标层下面一层是 E, F 。使用三明治集成的具体步骤如下：

- 首先对目标层上面的一层使用由顶向下的集成策略，因此测试 A ，使用桩代替 B, C, D ；
- 其次对目标层下面的一层使用自底向上的集成策略，因此测试 E, F ，使用驱动代替 B, D ；
- 其三，把目标层下面的一层与目标层集成，因此测试 $(B, E), (D, F)$ ，使用驱动代替 A ；
- 最后，把三层集成到一起，因此测试 (A, B, C, D, E, F)

具体如图 7-8 所示

注：在进行三明治集成时，需要记住尽可能减少驱动和桩模块的数量。例如在上面的例子中，我们选取使用由目标层下面一层与目标层先集成，而不是使用目标层上面一层与目标层先集成，因为这样可以减少桩模块的设计。

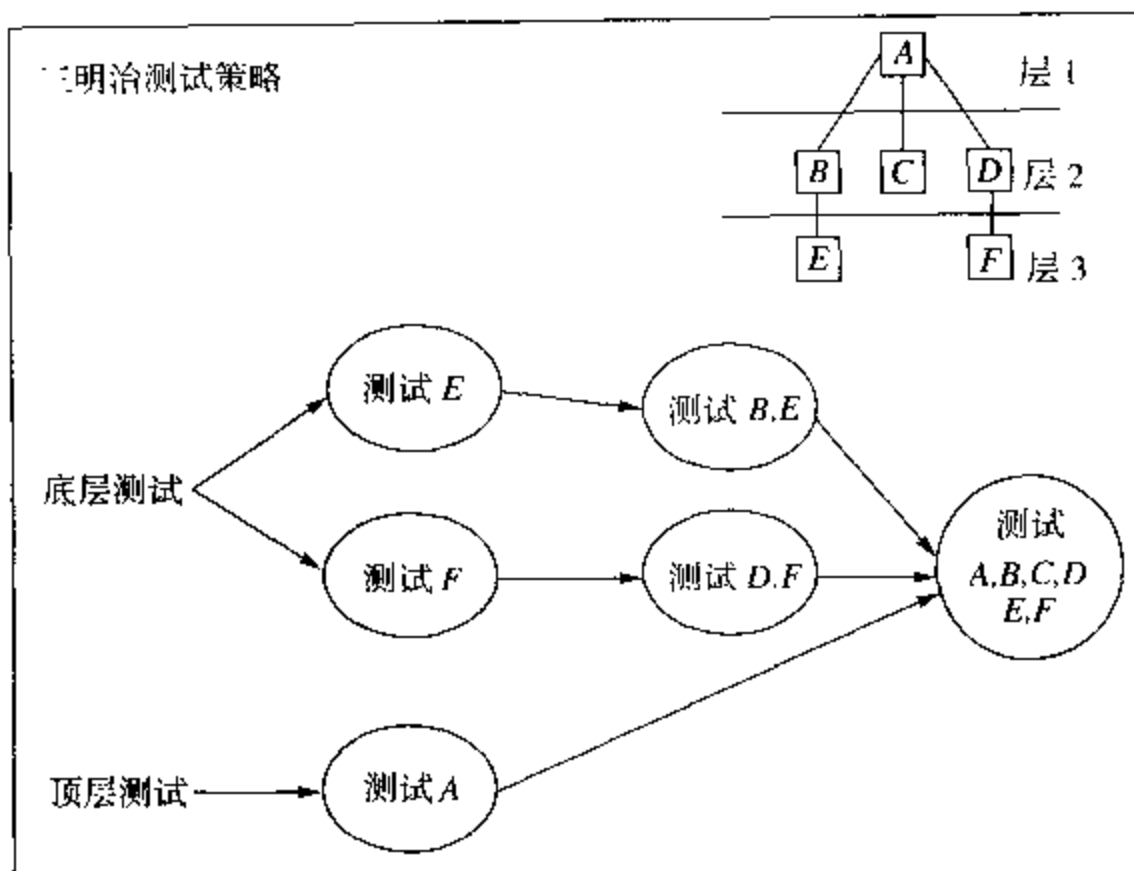


图 7-8 三明治测试策略

4. 优点

集合了由顶向下和自底向上的两种集成策略的优点。

5. 缺点

中间层在被集成前测试不充分。

6. 适用范围

大部分软件开发项目都可以使用这种集成策略。

7.2.5 修改过的三明治集成

1. 目的

弥补三明治集成不能充分测试中间层的缺点，尽可能提高测试的并行性。

2. 介绍

略。

3. 策略

该方法的具体步骤如下：

- 并行测试目标层，目标层上面一层，目标层下面一层。其中对目标层上面一层使用自顶向下集成策略，目标层下面一层使用自底向上集成策略，对目标层使用独立测试策略（需要驱动和桩）；

- 并行测试目标层与目标层上面一层的集成和目标层与目标层下面一层的集成。具体如图 7-9 所示。

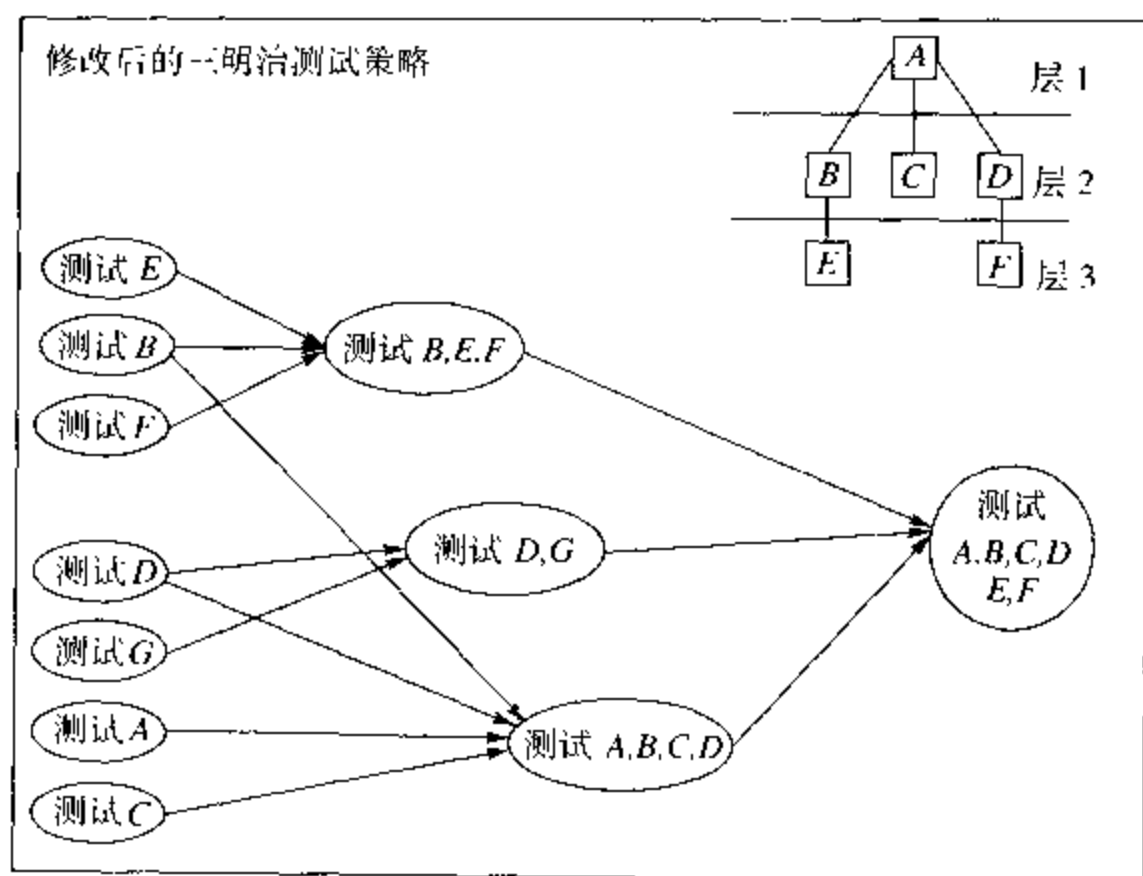


图 7-9 修改后的三明治集成

4. 优点

- 具有三明治集成的所有优点，且对中间层能够尽早进行比较充分的测试；
- 该策略的并行度比较高。

5. 缺点

中间层如果选取不恰当，可能会有比较大的驱动模块和桩模块的工作量。

6. 适用范围

大多数软件开发项目。

7.2.6 基于集成

1. 目的

结合自顶向下，自底向上和大爆炸集成的元素，以验证紧密耦合的子系统间的互操作性。

2. 介绍

基于集成 (Backbone Integration) 是由 Beizer^[26] 提出来的。

在很多系统中，尤其在嵌入式系统中，一般可以划分成两个部分：内核部分 (基干部分) 和外围应用部分。这两部分经常会被不同的项目组并行开发，具有如下特点：

- 内核部分提供了系统最基本的功能和服务；
- 外围应用以内核为基础，不能脱离内核而独自使用；
- 内核具有很高的耦合性，并且相当复杂，试图设置其桩模块会是相当困难且成本很高的事情；
- 高层控制层可以使用较高层或中间层做桩来进行测试；
- 中间层是有一些模块组构成，这些模块组内具有较高的耦合性，而模块组之间耦合较松散。

3. 策略

基于集成策略首先应识别应用的控制组件部分、基干部分和应用子系统部分。测试的顺序是基于这个分析结果的。其具体的测试步骤大致可以表述如下：

- 对基干中的每个模块进行单独的、充分的测试，必要时使用驱动器和桩模块；
- 对基干中所有的模块进行大爆炸集成，形成基干子系统，并使用一个驱动模块检查经过大爆炸的基干；
- 对应用的控制子系统进行自顶向下的集成；
- 把基干和控制子系统进行集成，重新构造控制子系统；
- 对各应用子系统采用自底向上的集成策略；
- 集成基干子系统、控制子系统和各应用子系统形成整个系统。

4. 优点

具有三明治集成的优点，更适合于大型复杂项目的集成。

5. 缺点

- 必须对系统的结构和相互依存性进行仔细的分析；
- 必须开发桩模块和驱动模块，并且由于被测系统的复杂性导致这些模块开发工作量的加大，可以通过复用技术在一定程度上降低成本；
- 由于局部采用了大爆炸的策略，因此有些接口可能测试不完整。

6. 使用范围

基于集成策略比较适合于大型复杂项目，一般来说，对于具有如下特定的项目可以优先考虑：

- 具有多层协议的嵌入式系统开发；
- 操作系统产品。

7.2.7 分层集成

1. 目的

通过增量式集成的方法验证一个具有层次体系结构的应用系统的稳定性和可互操作性。

2. 介绍

分层模型在通信系统中是很常见的。分层集成(Layers Integration)就是针对这个特点使用的一种集成策略。系统的层次划分可以通过逻辑的或物理的手段进行。在逻辑上,一般通过功能把系统划分成不同功能层次的子系统,子系统内部具有较高的耦合性,子系统间的关系具有线性层次关系;在物理上,可以根据不同单板内的系统划分为不同的硬件子系统,各硬件子系统之间根据连接具有线性层次关系。如果各层之间有拓扑网络关系,则不适合使用该集成方法。

3. 策略

分层集成的具体步骤大致如下:

- 划分系统的层次;
- 确定每个层次内部的集成策略,该策略可以使用大爆炸集成、自顶向下集成、自底向上集成和三明治集成中的任何一种策略。一般对于顶层可能还有第二层的内部采用自顶向下的集成策略;对于中间层采用自底向上的集成策略;对于底层主要进行单独测试;
- 确定层次间的集成策略,该策略可以使用大爆炸集成、自顶向下集成、自底向上集成和三明治集成中的任何一种策略。

图7-10和图7-11给出了一个分层集成的示意图。图中,层之间采用了由顶向下的集成策略。

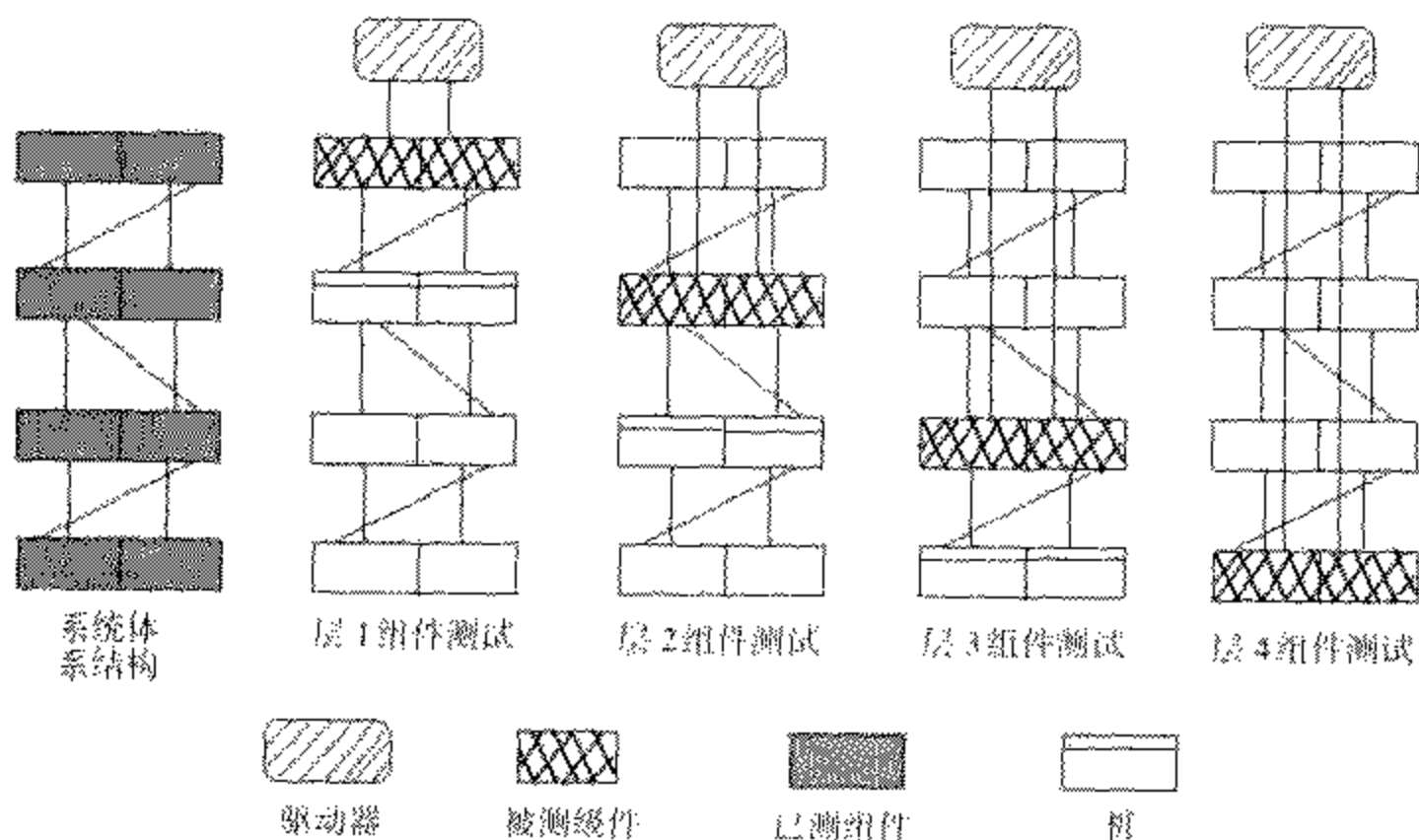


图7-10 层次内集成

4. 优点

其优缺点与其使用的层间集成测试策略类似。

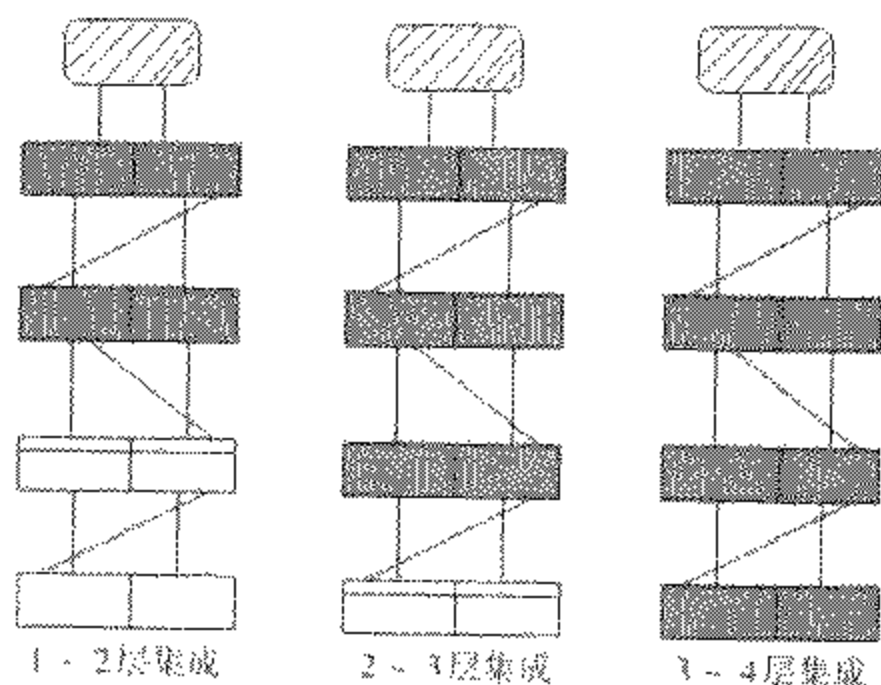


图 7-11 层间集成

5. 缺点

同上。

6. 使用范围

有明显线性层次关系的产品系统。

7.2.8 基于功能的集成

1. 目的

采用增值的方法，尽早地验证系统关键功能。

2. 介绍

在开发过程中，如果能尽早看到系统的主要功能被实现，这对整个团队的士气是一个极大的鼓舞。基于功能的集成(Function-Based Integration)是从功能角度出发，按照功能的关键程度对模块的集成顺序进行组织。

3. 策略

基于功能的集成策略大致步骤如下：

- 确定功能的优先级别；
- 分析优先级最高的功能路径，把该路径上的所有模块集成到一起，必要时使用驱动模块和桩模块；
- 增加一个关键功能，继续步骤2，直到所有模块都被集成到被测系统中。

注：为了提高集成的有效性，在执行步骤2时，会优先考虑关键功能的正常路径(尽可能覆盖最少模块集)，然后逐步考虑该功能的异常路径，并增加相应的异常处理模块。对于步骤3，一般在选择下一个关键功能时，需要看该功能的增加会不会引起新模块的加入，如果不会引起新模块的加入，那么可以考虑选择再下一个关键功能，因为这里的测试

是要做到所有模块的覆盖，而不是功能的覆盖或路径的覆盖。

4. 优点

- 采用该方法，可以尽快地看到关键功能的实现，并验证关键功能的正确性；
- 由于该方法在验证某个功能时，可能会同时加入多个模块，因此在进度上比自底向上、自顶向下或三明治集成要短；
- 接口的覆盖使用的测试用例比较少；
- 可以减少驱动模块的开发，原因与自顶向下的集成策略类似。

5. 缺点

- 对于复杂系统，功能之间的相互关联性可能是错综复杂并难以分析的；
- 对有些接口的测试不充分，会丢失许多接口错误；
- 一些初始的集成需要使用桩模块；
- 可能会有比较大的冗余测试。

6. 使用范围

- 关键功能具有较大风险的产品；
- 技术探索型的项目，其功能的实现远比质量更关键；
- 对于功能实现没有把握的产品。

7.2.9 高频集成

1. 目的

频繁地将新代码加入到一个已经稳定的基线(Baseline)中，以免集成故障难以发现，同时控制可能出现的基线偏差。

2. 介绍

快速迭代式开发或增量式开发可能会导致功能的遗漏或冲突。其最初的起始可能是一个最低功能限度的集成包，随着新代码的迅速开发并加入到系统中，需要验证扩大后的系统是否是稳定的，并且功能是正确的，如果这些问题遗留到最后再检查，其成本可能将很巨大。高频集成(High-frequency Integration)就是采用的这个策略。使用高频集成需要具备以下条件：

- 可以获得一个稳定的增量，并且已完成的某子系统已被验证没有问题；
- 大部分有意义的功能增加可以在一个固定的频率间隔内获得，比如通过每日创建来获得；
- 测试包和代码并行开发，并且始终维护的是最新的版本；
- 必须使用自动化，例如采用 GUI 的捕获/回放工具；
- 必须使用配置管理工具，否则版本的增量将无法控制。

3. 策略

高频集成有 3 个主要的步骤，具体如下。

(1) 开发人员完成要提供的代码的增量部分，同时测试人员完成相关的测试包，具体包括：

- 编写或修改代码；
- 编写或修改对相应代码的测试包；
- 对新增或修改过的代码进行代码走读，检视和评审；
- 对测试包进行代码走读，检视和评审；
- 对修改或新增的组件进行静态分析(如：PC-LINT 检查等，一般通过工具来完成)；
- 对代码进行创建工作(Build)；
- 在新的创建上运行测试包，包括使用类似内存检测工具，性能检测工具进行跟踪检查；
- 当组件通过所有测试时，将已修改过的测试包提交到集成测试部门。

(2) 集成测试人员将开发人员修改或增加的组件集中起来形成一个新的集成体，并且在上面运行集成后的测试包。具体工作包括：

- 在一个既定的规定期限内，负责集成的人员终止接受任何增量，并形成一个新系统的基线；
- 进行创建工作，并在上面运行测试包。这个测试将包括冒烟测试，新开发的测试。如果时间允许应尽可能多地运行测试。

(3) 评价结果。“在建造过程中，纠正错误是项目中优先级最高的”^[132]。

在高频集成中，必须切实有效地解决下列问题：

- 谁维护现有的集成测试包？
- 创建的周期是多长？
- 谁来进行创建工作？谁来做集成测试？在什么情况下进行？
- 当创建失败或测试没有通过时，采取什么后续的动作？系统将退回到哪个版本？问题定位和纠正的任务分配给哪个人或哪个组织？
- 如何进行自动化？

4. 优点

- 由于在该策略中，开发维护源代码和测试包具有同等的重要性，这对有效防止错误非常有帮助；
- 严重错误、遗漏和不正确的假设经常能被较早地揭示；
- 当错误产生时，其最可能存在于新增加或修改的代码中，这对调试非常有利；
- 整个开发组集中于生产一个运转的系统，而不是用于实际工作的一个系统。这有助于开发人员尽早能够看到一个可运行的系统，并提高士气；
- 对桩代码的需要不是必须的，这可以避免编写和维护容易损坏的测试代码；
- 开发和集成可以并行进行。

5. 缺点

- 测试包可能会过于简单，并因此难以发现有价值的问题；
- 在刚开始的几个周期可能不易于平稳地进行集成；
- 如果没有对高频集成建立适当的标准，一长串成功的集成可能导致不应有的可信度，这往往会使风险增加。

6. 使用范围

采用迭代(或增量)过程模型开发的产品。

7.2.10 基于进度的集成

1. 目的

尽可能早地进行集成测试，提高开发与集成的并行性，有效地缩短进度。

2. 介绍

进度压力是每个软件开发项目都会遇到的问题。为了完成进度，很多项目往往牺牲了部分质量，并且加班加点地疲劳工作。基于进度的集成(Schedule-Based Integration)就是在兼顾进度和质量两者之间寻找了一个均衡点。该集成的一个最基本的策略就是把最早可获得的代码拿来立即进行集成，必要时开发桩模块和驱动模块，在最大限度上保持与开发的并行性，从而缩短了项目集成的时间。

3. 策略

略。

4. 优点

- 具有比较高的并行度；
- 能够有效缩短项目开发的进度。

5. 缺点

- 可能最早拿到的模块之间缺乏整体性，只能进行独立的集成，导致许多接口必须等到后期才能验证，但此时系统可能已经很复杂，往往无法发现有效的接口问题；
- 桩模块和驱动模块的工作量可能会变得很庞大；
- 由于进度的原因，模块可能很不稳定且会不断变动，导致测试的重复和浪费。

6. 使用范围

进度优先级高于质量的项目。

7.2.11 基于风险的集成

1. 目的

在第一时间内验证高危模块间的接口，从而保证系统的稳定性。

2. 介绍

基于风险的集成(Risk-Based Integration)是基于这样一个假设，既系统风险最高的模块间的集成往往是错误集中的地方，因此尽早地验证这些接口有助于加速系统的稳定，从而增加对系统的信心。该方法与基于功能的集成有一定的相通之处，可以结合使用。

3. 策略

此处略。

4. 优点

最具有风险的模块最早进行验证，有助于系统的快速稳定。

5. 缺点

需要对各组件的风险有一个清晰的分析。

6. 使用范围

项目有些模块具有较大的风险，且没有信心。

7.2.12 基于事件(消息)的集成

1. 目的

从验证消息路径的正确性出发，渐增式地把系统集成到一起，从而验证系统的稳定性。

2. 介绍

对于许多基于状态机的系统来说(例如，嵌入式系统、面向对象系统)，其工作原理是基于状态变迁，内部模块间的接口主要是通过消息来完成的。因此验证消息路径的正确性对于这类系统具有比较重要的位置，基于消息/事件/线程的集成(Message-Based/Event-Based/Thread-Based Integration)就是针对这一特点而设计的一种策略。

3. 策略

- 从系统的外部看，分析系统可能输入的消息集；
- 选取一条消息，分析其穿越的模块；

- 集成这些模块进行消息接口测试；
- 选取下一条消息，重复步骤2和3，直到所有模块都被集成到系统中。

注：消息的选取可以从3个角度考虑。

- 消息的重要性；尽早验证重要的消息路径；
- 消息路径的长度；为了能有效验证接口的完整性和正确性，尽可能选取路径较短的消息；
- 新的消息的选择是否能够使得新的模块被加入到系统中。

上面提到的模块包括类和进程。

4. 优点

参考基于功能的集成。

5. 缺点

参考基于功能的集成。

6. 使用范围

- 面向对象系统；
- 基于有限状态机的嵌入式系统。

7.2.13 基于使用的集成

1. 目的

针对面向对象系统，通过类之间的使用关系来集成系统，从而验证系统的稳定性。

2. 介绍

在一个面向对象系统中，存在一些独立的类和一些相互耦合的类。基于使用的集成(Use-Based Integration)从分析类之间的依赖关系出发，通过从最小依赖关系的类开始集成，逐步扩大到有依赖关系的类，最后集成到整个系统。通过该集成方法，可以验证类之间接口的正确性。该方法可以和分层集成或其他集成策略结合使用。

3. 策略

- 划分类之间的耦合关系；
- 首先测试独立的类；
- 其次测试使用一些服务器类的类；
- 最后逐步增加具有依赖性的类(既使用独立类的类)，直到整个系统被集成到一起。

4. 优点

参考由底向上的集成测试策略。

5. 缺点

参考由底向上的集成测试策略。

6. 使用范围

面向对象系统。

7.2.14 客户/服务器的集成

1. 目的

验证客户和服务器之间交互的稳定性。

2. 介绍

对于和单独的服务器组件进行松散耦合的客户端组件，可以使用客户/服务器集成 (Client/Server Integration) 来完成。和自顶向下的策略不同，在这个模型中，不存在单独的控制轨迹。服务器对客户的信息做出反应，客户对来自系统环境的信息做出反应。每个系统组件都有其自己的控制策略。

3. 策略

- 单独测试每个客户端和服务端，必要时使用驱动和桩；
- 把第一个客户端(客户端组)与服务端进行集成；
- 把下一个客户端(客户端组)与步骤2完成的系统进行集成；
- 重复步骤3直到所有客户端都被加入到系统中。

4. 优点

- 避免了大爆炸集成的风险；
- 集成次序没有大的约束，可以结合风险或功能优先级进行；
- 有利于复用和扩充；
- 支持可控制和可重复的测试。

5. 缺点

驱动器和桩的开发成本可能会比较高。

6. 使用范围

客户/服务器结构的系统。

7.2.15 分布式集成

1. 目的

验证松散耦合的同级组件之间交互的稳定性。

2. 介绍

被测系统包括许多并发运行，且没有专门控制轨迹的组件，以及没有专门服务器层，它们构成了一个分布式系统。分布式集成(Distributed Services Integration)就是针对这个特点设计的一个策略。

3. 策略

集成测试一个分布式系统最关心的是验证远程主机之间的接口是否具有最低限度的可操作性。在一个分布式系统中，构造测试包是比较困难的，最坏的情况是每个节点之间都具有比较强的连接(强连接图)。由于没有专门的控制轨迹，所以接口的测试顺序可以有很多种，下面提供一些选择：

- 风险驱动(Risk Driven)。即从最可能出现问题的接口和组件开始进行集成。
- 反风险驱动(Risk Averse Driven)。即从最不可能出现问题的接口和开始进行集成。
- 依赖性驱动(Dependency Driven)。从可以被单独测试或依赖性最小的组件接口开始集成。
- 优先驱动(Priority Driven)。从功能或性能优先级高的组件和接口开始进行集成。

4. 优点

类似客户/服务器集成。

5. 缺点

类似客户/服务器集成。

6. 使用范围

分布式系统。

7.3 集成测试分析

要做好集成测试，必须加强集成测试的分析工作。类似于概要设计对详细设计的作用，集成测试分析直接指导了集成测试用例的设计，并且在整个集成测试过程中占据了最关键的地位。

集成测试的分析可以着重在以下几个方面进行考虑。

7.3.1 体系结构分析

体系结构的分析需要从两个角度出发，首先从需求的跟踪实现出发，划分出系统实现上的结构层次图形，类似于图 7-1，图 7-2，图 7-3。这个结构图对集成的层次考虑是有帮

助的。

其次需要划分系统组件之间的依赖关系图，如图 7-12 所示。通过该图的分析，需要划分出集成测试的粒度，即基础模块的大小。集成测试模块的划分是一件头疼的事，模块划分到底要多大？是否需要设计驱动和桩？该模块的划分能否有效降低消息接口的复杂性？接口是否充分等？模块划分得好，可以极大提高集成测试的效率，反之则会降低集成测试的质量。关于模块的划分请参考下一节。

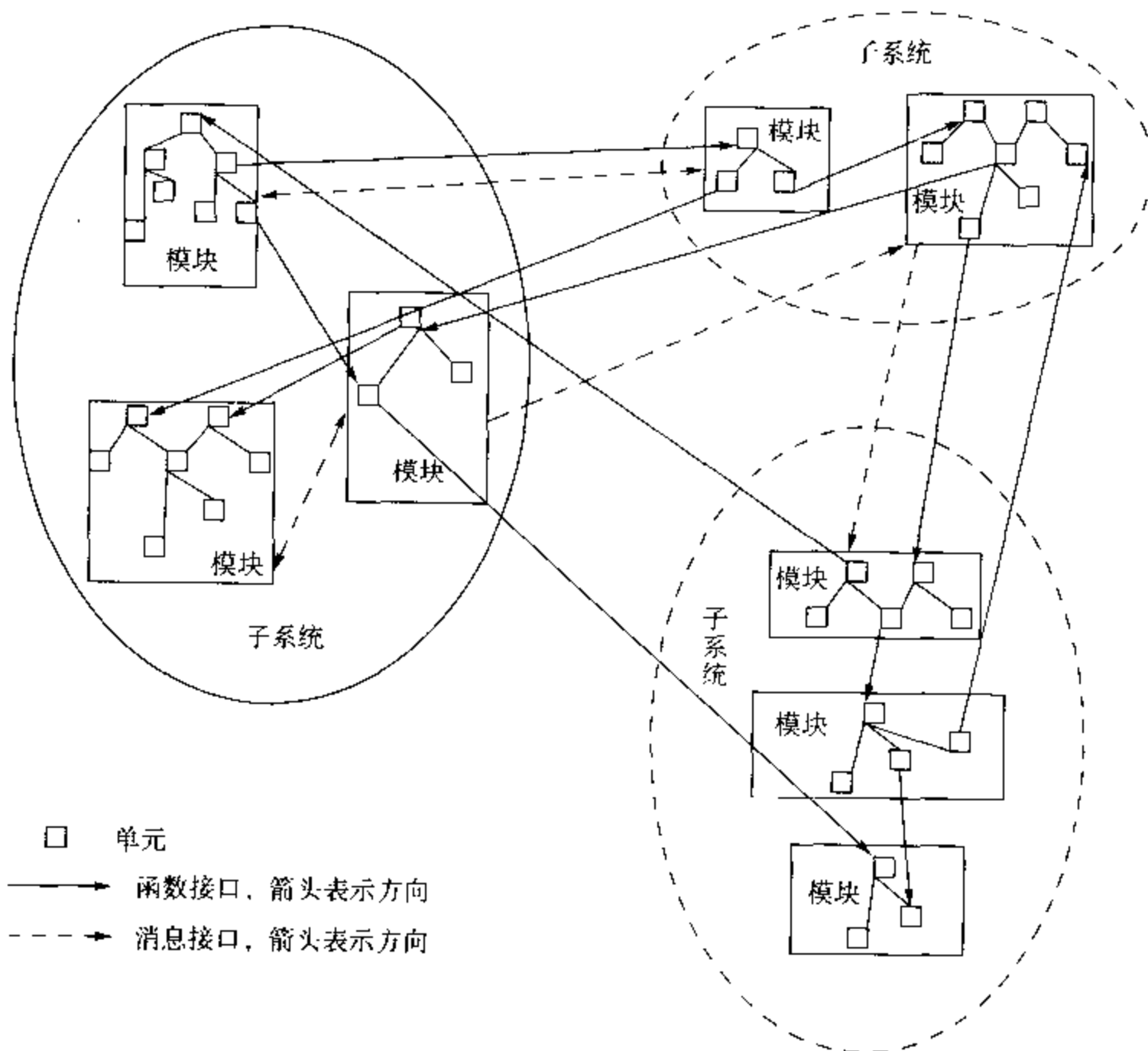


图 7-12 系统依赖关系示意图

体系结构的分析还为集成测试策略的选择提供了思路。

7.3.2 模块分析

模块分析是集成测试分析最重要的工作之一。模块划分的好坏直接影响了集成测试的工作量、进度以及质量。因此需要慎重对待模块的分析。

一般模块划分可以从下面几个角度出发进行考虑：

- 本次测试主要希望测试哪个模块；
- 这个模块与哪几个模块有最密切关系，可以一、二、三、四按照密切程度排队；

- 把该模块与关系最密切的模块首先集成在一起；
- 这时再考虑这样划分后的外围模块，这些模块与被集成模块之间的消息流是否容易模拟，是否方便控制。

一个合理的集成模块划分应该满足以下几点：

- 被集成的几个子模块关系紧密；
- 外围模块便于屏蔽，外围模块与集成模块之间没有太多、太频繁的调用关系，被集成模块没有采用类似 POST_MESSAGE 等调用函数的方式调用外围模块的情况。如果实在无法避免，就不得不考虑编写桩函数或桩模块，以代替被屏蔽部分的功能；
- 模拟外围模块发往被集成模块的消息便于构造、修改；
- 外围模块发往被测试模块的消息能够模拟大部分实际环境的情况。

在软件工程中有一条事实上的原则，即 2/8 原则，该原则对测试同样起作用。从测试的实践发现，测试中发现的错误 80% 很可能起源于程序模块中的 20%^[6]。并且经验也告诉我们，很多错误报告常常可以在同一个模块中被追踪到，并且统计数据表明一段程序中已经发现的错误数目往往和尚未发现的错误数成正比。例如，在 IBM OS/370 操作系统中，用户发现的全部错误的 47% 只与该系统中 4% 的模块有关。对这样的一些模块，我们称之为易错模块或高危模块。一般我们可以把系统中的模块划分成 3 个等级：高危模块（这是集成测试需要关注的关键模块）、一般模块和低危模块（如果时间不允许，往往会减少或忽略这部分模块的集成，同时可能会对这类模块直接采用大爆炸式集成策略）。所以，划分集成测试对象模块时，首先应该判断系统中哪些是关键模块。

一个关键模块具有一个或多个下列特性：

- 和多个软件需求有关，或与关键功能相关；
- 处于程序控制结构的顶层；
- 本身是复杂的或者是容易出错的；
- 含有确定性的性能需求；
- 被频繁使用的模块（这部分模块不一定会出错，但可能会是性能的瓶颈，并且这类模块一旦出错，影响会比较大）。

在实际操作时，可以通过以下途径来分析关键模块：

- 尽可能和开发人员多讨论讨论，听听他们的意见，一般开发人员对哪些模块是关键模块，心理比较清楚；
- 通过使用静态分析工具来分析系统各模块，寻找高内聚的模块、被频繁调用的模块或处于控制顶层的模块；
- 根据需求跟踪表来分析关键模块，一般与关键功能或关键接口相关的模块都是比较关键的。同时与一些特殊需求相关的模块也需要特别关注；
- 对于一个维护型的项目，可以根据以往的历史经验来判定，这包括产品历史缺陷的分析；
- 对于一个新的产品，可以根据开发过程前期包括文档的检视、代码的走读、单元测试发现的问题进行分析，并因此确定可能风险最大的模块。

7.3.3 接口分析

我们知道,集成测试的重点就是测试接口的功能性、可靠性、安全性、完整性、稳定性等多个方面。因此我们必须对被测对象的接口进行详细地分析。这些分析包括接口的划分、接口的分类和穿越接口的数据分析。

1. 接口的划分

接口的划分是以概要设计为基础的,其方法与相关的结构设计技术类似。一般可以通过下面几个步骤来完成:

- 确定系统的边界、子系统的边界和模块的边界,参考图 7-12;
- 确定模块内部的接口;
- 确定子系统内模块间接口;
- 确定子系统间接口;
- 确定系统与操作系统的接口;
- 确定系统与硬件的接口;
- 确定系统与第三方软件的接口。

2. 接口的分类

在实际环境中,我们会遇到各种各样的接口,关于接口的分类也有很多种,但总的来说可以划分为以下两个大类。

系统内接口 系统内部各模块交互的接口,这是集成测试的重点;

系统外接口 外部系统(包括人、硬件和软件)对系统交互的接口,这类测试一般会延续到系统测试阶段来完成。

在此我们仅定义系统内的接口,主要包括以下几种。

函数接口 通过函数的调用和被调用关系来确定。关于函数接口的集成测试比较成熟,在 7.2 节中提到的大部分集成策略都可以应用到这类接口上。

消息接口 消息接口在面向对象系统和嵌入式系统中是很普遍的。在这种接口下,软件模块间并不直接发生联系,而是通过消息包(遵循接口协议)发生关系,常见的例子是整个系统共用一个或多个消息包队列,由操作系统进行消息包调度,取出位于消息队列头的消息并调用该消息的处理模块的处理函数。该处理模块,其核心通常是一个被动的有限状态机模型,根据消息内容和自身状态做出反应,通常是完成状态迁移并将发往另一个模块的消息放到消息队列中去。对于这种接口的测试我们往往采用工具来模拟,在集成策略上可以使用 7.2 节中支持有限状态机模型的集成策略。

类接口 面向对象系统中,类接口是基本的接口。类接口一般可以通过继承、参数类、不同类方法调用等策略来实现。对于这类系统,传统的集成策略比较难以应用。一般可以使用基于线程的集成和基于使用的集成策略。由于面向对象软件分析可以使用 UML,因此很多人已经开始研究基于 UML 的集成测试策略,并已经得到了实际的应用^{[133][134]}。由于面向对象系统的集成分析已经超出了本书的范围,因此就不详细描述了,如果想了解

这方面技术的话,可以参考附录 E^{[127][133][134][135][136][137][138][139][140][141]}。

其他接口 其他类型接口包括全局变量、配置表、注册信息、中断等。这类接口具有一定的隐蔽性,往往测试人员会忽略这部分接口。从笔者接触到的一些产品测试来看,这类接口经常是测试不充分的。这类接口的测试需要借助一定的自动化工具。

3. 接口数据分析

接口数据分析就是要分析穿越接口的数据。从这些数据的分析过程中可以直接产生测试用例。接口类型的不同,在数据分析上略有不同。

对于函数接口,我们需要关注穿越函数接口的参数个数、参数属性(参数类型,输入输出属性)、参数的顺序、参数的等价类情况、参数的边界值情况等。如果需要的话还要考虑它们的组合情况。

关于消息接口,需要分析消息的类型、消息的域、域的顺序、域的属性、域的取值范围、可能的异常值等。如果需要的话,也要考虑它们的组合情况。

关于类接口,我们需要对类的属性进行分析,一般重点在于对公共属性和保护属性进行分析,必要时会对部分私有属性进行分析。分析的方法包括等价类划分和边界值分析。

对于其他类接口的分析,我们需要分析其读写属性、并发性、等价类和边界值。特别对于配置文件这类接口,其涉及到的数据变化量是极其庞大的,在分析这类数据时,可能需要结合一定的约束条件,尽可能减少用例的数量。

7.3.4 风险分析

风险分析是贯穿于整个过程当中的,必须始终正确对待风险,不能期望风险会被偶然绕过去。很多有关软件工程和项目管理方面的书都提到了风险分析^{[6][142][143][144][145][146]},大家可以认真学习学习,这是非常有帮助的。

一般风险分析包含3个阶段:风险识别、风险评估和风险处理。

(1) 风险识别。识别风险的方法可以依靠观察,掌握有关的知识,调查研究,参考相关资料和经验数据,听取专家意见等。一般有头脑风暴法和 DELPHI 法。

(2) 风险评估。对已识别的风险要进行估计,主要任务是确定风险发生的概率和后果。

(3) 风险处理。一旦风险被识别、评估以及风险量被确定之后,就要考虑各种风险的处理方法。针对集成测试过程来说,一般有3种方法:

- 风险控制:包括主动采取措施避免风险、消灭风险、中和风险,或一旦风险发生,即采取紧急应急方案,力争将损失减少到最低限度;
- 风险自留:如果风险量被确认不大,不超过集成测试应急资源时,可以将该风险留在项目组中;
- 风险转移:将风险转移给另一方或其他测试阶段。

在集成测试中,常见的风险包括:

- 技术风险

如测试人员对集成测试技术掌握比较薄弱或没有类似产品的集成测试经验;产品缺乏

相关技术文档尤其是对接口描述稳定的缺乏；测试人员缺乏产品背景知识；测试人员对相关集成测试工具使用不了解等等。

- 人员风险

如人员变动频繁，人员到位不及时，缺乏有经验的老员工等。

- 物料仪器

测试环境如电脑、单板或其他硬件风险；物料仪器申购风险；测试工具无法及时到位风险等。

- 管理风险

版本计划更改风险；人员、时间计划变更风险；缺乏有效配置管理；过程失控；开发进度延迟等。

- 市场风险

市场需求更改；市场供货时间更改等。

7.3.5 可测试性分析

系统的可测试性分析应当在项目开始时作为一项需求提出来，并设计到系统中去。在集成测试阶段，分析可测试性主要是为了平衡随着集成范围的增加而导致的可测试性下降。对于一个接口不可测的系统，集成测试的实现是相当困难的，这会导致大量测试代码的添加或接口测试工具的开发。尽可能早地分析接口的可测试性，提前为测试的实现做好准备。

7.3.6 集成测试策略分析

集成测试策略分析主要根据被测对象选择合适的集成策略。一般来说，一个好的集成测试策略应该具有以下特点：

- 能够对被测对象进行比较充分的测试，尤其是对关键模块；
- 能够使模块与接口的划分清晰明了，尽可能地减小后继操作难度，同时使需要做的辅助工作量最小；
- 整体工作量对于投入测试的资源来说大致相当，参加测试的人力、环境、时间等资源能够得到充分利用。

在7.2节我们介绍了很多集成测试策略，而在实际使用时，一般不会选择所有的策略，也不会只选择一种策略，通常需要根据对被测试对象的体系结构分析、模块分析、接口分析、风险分析、可测试性分析、人力分析、测试环境分析以及测试进度分析来综合选择多种测试策略。尽可能花费最小的成本，取得最大的测试效果。

7.3.7 常见的集成测试故障

了解集成测试中常见的故障，对于集成测试数据的分析是有帮助的。一般的接口错误包括^[127]：

- 配置/版本控制错误;
- 遗漏、重叠或冲突的函数;
- 文件或数据库使用不正确的或不一致的数据结构;
- 文件或数据库使用冲突的数据视图/用法;
- 破坏全局存储或数据库数据的完整性;
- 由于编码错误或未预料到的运行时绑定导致的错误方法调用;
- 客户发送违反服务器前提条件的消息;
- 客户发送违反服务器顺序约束的消息;
- 错误的对象和消息的绑定(多态中经常发生);
- 错误参数或不正确的参数值;
- 由不正确地内存管理分配/收回引起的失败;
- 不正确地使用虚拟机、ORB 或 OS 服务;
- IUT 试图使用目标环境的服务,而该服务对目标环境的指定版本是已经过时或不向上兼容的;
- IUT 试图使用目标环境的新服务,而该目标环境当前的版本不支持该服务;
- 组件之间的冲突,例如当进程 Y 运行时,线程 X 就会崩溃;
- 资源竞争:目标环境不能分配象征性装载所需要的资源,例如:一个用例可能打开 6 个窗口,但是 IUT 在打开 5 个以后就崩溃了。

7.4 集成测试用例设计思路

就如本章开头所分析的,集成测试是介于白盒测试和黑盒测试之间的灰盒测试,因此在该测试的用例设计方法中会综合使用两类测试中的测试分析方法。一般经过集成测试分析之后,测试用例的大致轮廓已经确定,集成测试用例设计的基本要求就是要充分保证其正确性,保证其能无误地完成测试项既定的测试目标。在此我们根据单元测试用例设计的思路,来分析集成测试的用例设计(其实不管做哪个级别的测试,一般都脱离不了这些思路,不同的是思维的重点和覆盖的范围)。在实际操作时,可能需要综合这些方法。

7.4.1 为系统运行设计用例

集成测试的第一个需要关注问题是接口能用,并且不会阻塞后续的集成测试执行。因此可以根据这个原则设计一些基本功能的测试用例来进行最低限度的验证。

可使用的测试分析技术:

- 规范导出法;
- 等价类划分。

7.4.2 为正向测试设计用例

假设我们的过程是良好的,接口设计及模块功能设计需求是明确和无误的,那么作为集成测试的一个重点就是需要验证这些接口需求和集成后的模块功能需求被正确无误地满足了。基于这个原则,我们可以直接根据概要设计文档导出相关的用例。

可使用的测试分析技术:

- 规范导出法;
- 输入域测试;
- 输出域覆盖;
- 等价类划分;
- 状态转换测试。

7.4.3 为逆向测试设计用例

在集成测试中的逆向测试包括分析被测接口有否实现规格并没有要它实现的功能,规格中可能出现的接口遗漏或接口定义错误,分析可能出现的接口异常情况,包括接口数据本身的错误,接口数据顺序错误等。有时,经过接口的数据量是相当庞大的,例如在电信软件中,一些协议消息,动辄就包括了几十、上百个 IE (Information Element)。考虑所有 IE 的异常情况,甚至异常组合几乎是不可能的。因此在这--点上还需要基于一定的条件约束,包括分析一些关键的 IE、不可能的组合情况、需要考虑的组合等。

对于面向对象系统和基于有限状态机的系统还需要考虑可能出现的状态异常,包括:丢失的或不正确的状态转换;一个有效的消息被忽略;不可预测的行为;一个可能的潜行路径;一个不期望的消息引起的失败;接受没有定义的消息等。

可使用的测试分析技术:

- 错误猜测法;
- 基于风险的测试;
- 基于故障的测试;
- 边界值分析;
- 特殊值测试;
- 状态转换测试。

7.4.4 为满足特殊需求设计用例

以前的观点中,类似安全性测试、性能测试、可靠性测试等主要在系统测试阶段进行,但是目前通过在软件设计过程中细化了这些特殊的需求,一些产品开始在模块设计文档中就明确了接口的安全性指标、性能指标等,这种情况下应尽早开展接口相对于这些特殊需求的测试,以便最终保证系统整体特殊需求的满足。

可使用的测试分析技术:规范导出法。

7.4.5 为高覆盖设计用例

不同于单元测试，在集成测试中最关注的覆盖是功能覆盖和接口覆盖。通过分析集成后模块的哪些功能没有被测试到，哪些接口没有被覆盖（尤其对于消息接口，所有可能的正常消息，异常消息都应当被验证）到来设计测试用例。

可使用的测试分析技术：

- 功能覆盖分析；
- 接口覆盖分析；

7.4.6 测试用例补充

我们不可能在一开始就 100% 地完成所有集成测试用例的设计，由于随着可能的功能增加、特性修改、缺陷修改等原因，我们还需要在集成测试的执行阶段不断更新和补充集成测试用例。

7.4.7 注意事项

成本、进度、质量是开发过程中需要均衡的 3 个点，同样在集成测试过程中，我们也需要考虑这三者之间的平衡。测试重点要突出，关键的接口必须被覆盖到，同时用例设计要考虑充分的可回归性和执行的自动化。

7.5 集成测试过程

一个测试从开发到执行遵循一个过程，这个过程的定义在不同的组织会有不同。在单元测试一章，我们介绍了 IEEE Std 1008-1987 定义的一个过程，该过程同样可以应用到集成测试。这次我们介绍 IEEE 的另一个过程标准 IEEE Std 1012-1998^[124]，该过程也可以应用到单元测试过程和系统测试过程中去。根据该标准，可以把集成测试划分为 4 个阶段：计划阶段、设计阶段、实现阶段、执行阶段（实施阶段）。

7.5.1 计划阶段

1. 时间安排

概要设计完成评审后大约一个星期。

2. 输入

- 需求规格说明书；
- 概要设计文档；

- 产品开发计划路标。

3. 入口条件

概要设计文档已经通过评审。

4. 活动步骤

- 确定被测试对象和测试范围；
- 评估集成测试被测试对象的数量及难度，即工作量；
- 确定角色分工和划分工作任务；
- 标识出测试各阶段的时间、任务、约束等条件；
- 考虑一定的风险分析及应急计划；
- 考虑和准备集成测试需要的测试工具、测试仪器、环境等资源；
- 考虑外部技术支援的力度和深度，以及相关培训安排；
- 定义测试完成标准。

5. 输出

集成测试计划。

6. 出口条件

集成测试计划通过概要设计阶段基线评审。

7.5.2 设计阶段

1. 时间安排

详细设计阶段开始。

2. 输入

- 需求规格说明书；
- 概要设计；
- 集成测试计划。

3. 入口条件

概要设计阶段基线通过评审。

4. 活动步骤

- 被测对象结构分析；
- 集成测试模块分析；
- 集成测试接口分析；
- 集成测试策略分析；

- 集成测试工具分析;
- 集成测试环境分析;
- 集成测试工作量估计和安排

5. 输出

集成测试设计(方案)。

6. 出口条件

集成测试设计通过详细设计基线评审。

7.5.3 实现阶段

1. 时间安排

在编码阶段开始后进行。

2. 输入

- 需求规格说明书;
- 概要设计;
- 集成测试计划;
- 集成测试设计。

3. 入口条件

详细设计阶段基线通过评审。

4. 活动步骤

- 集成测试用例设计;
- 集成测试规程设计;
- 集成测试代码设计(如果需要);
- 集成测试脚本(如果需要);
- 集成测试工具(如果需要)。

5. 输出

- 集成测试用例;
- 集成测试规程;
- 集成测试代码(如果有);
- 集成测试脚本(如果有);
- 集成测试工具(如果有)。

6. 出口条件

测试用例和测试规程通过编码阶段基线评审。

7.5.4 执行阶段

1. 时间安排

单元测试已经完成后就可以开始执行集成测试了。

2. 输入

- 需求规格说明书；
- 概要设计；
- 集成测试计划；
- 集成测试设计；
- 集成测试用例；
- 集成测试规程；
- 集成测试代码(如果有)；
- 集成测试脚本(如果有)；
- 集成测试工具(如果有)；
- 详细设计；
- 代码；
- 单元测试报告。

3. 入口条件

单元测试阶段已经通过基线化评审。

4. 活动步骤

- 执行集成测试用例；
- 回归集成测试用例；
- 撰写集成测试报告。

5. 输出

集成测试报告。

6. 出口条件

集成测试报告通过集成测试阶段基线评审。

7.6 集成测试环境

对于简单的系统来说(比如:单进程的软件),其集成测试环境与单元测试环境比较类似。然而,现在的系统越来越复杂,往往一个系统会分布在不同的硬件和软件平台上,需要所有分布在这些平台上的子系统或组件能够彼此集成为一个整体完成系统的功能。因此,对于这样的系统,其集成测试环境就要复杂多了。这时往往需要依赖一些商用测试工具或测试仪,可能还需要专门开发一些接口模拟工具。

在考虑集成测试环境时,可以从以下几个方面进行:

- 硬件环境

在集成测试时,尽可能考虑实际的环境。如果实际环境不可用时,考虑可替代的环境或在模拟环境下进行,如在模拟环境下使用,需要分析模拟环境与实际环境之间可能存在的差异。

- 操作系统环境

考虑不同机型使用的不同操作系统版本。对于实际环境可能使用的操作系统环境,尽可能都要被测试到。

- 数据库环境

数据库的选择要根据实际的需要,从性能、版本、容量等多方面考虑。

- 网络环境

一般的网络环境可以使用以太网。

图 7-13 是一个典型的集成测试环境示意图。

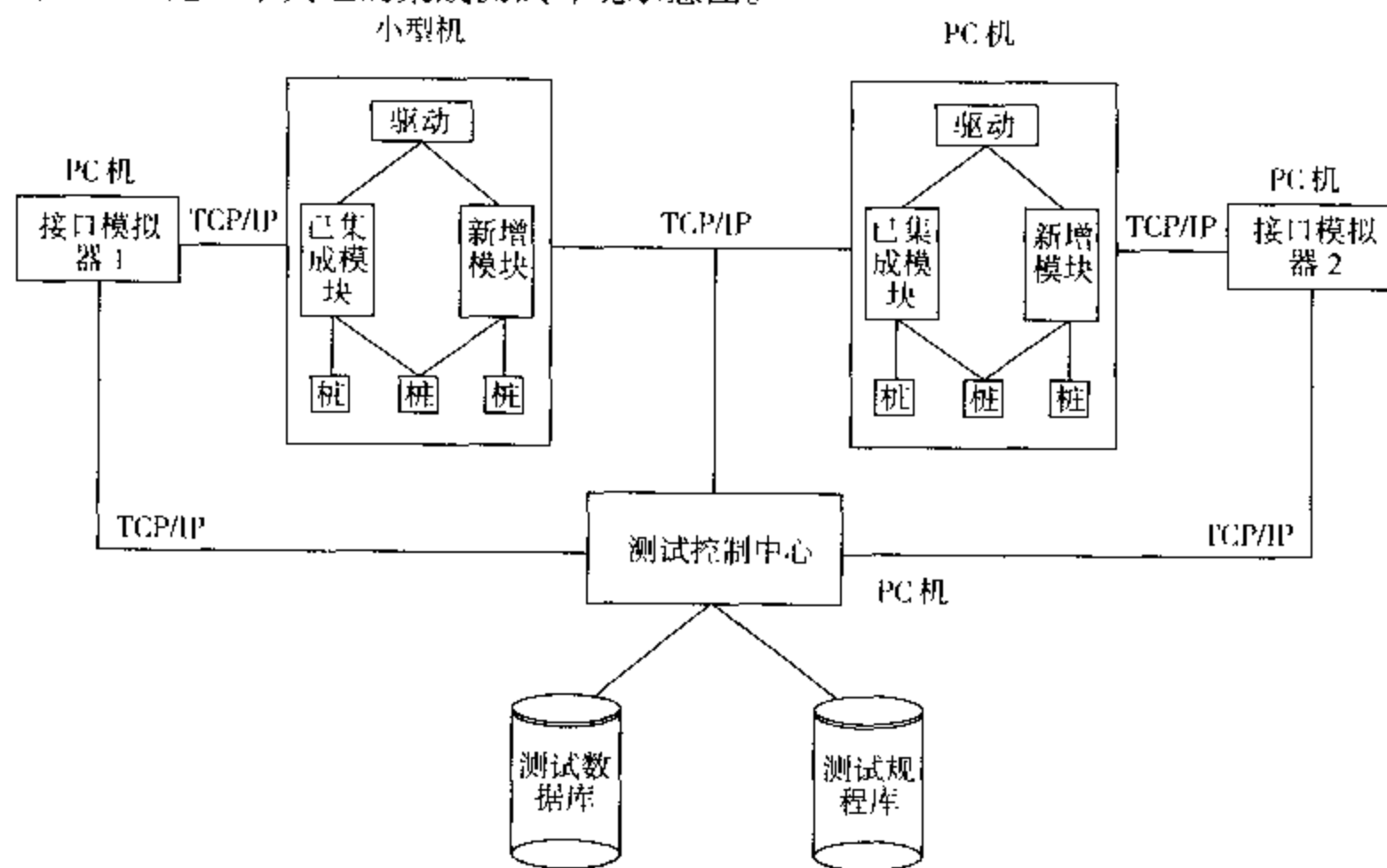


图 7-13 集成测试环境示意图

7.7 集成测试工具介绍

能够直接用于集成测试的测试工具并不多，一般来说，能够有效支持产品的集成测试工具大部分是自己开发的，而一些通用的商用测试工具由于要满足一定的通用性，因此在实际使用时往往功能有限，需要进行二次开发；另外对于一些专用的商用测试仪器或测试工具其使用范围相对都比较狭窄。

由于集成测试处于白盒和黑盒之间，因此用于单元测试的一些测试工具同样也可以使用于集成测试，但是需要处理的复杂程度将大大提高。

对于一些以界面为主的产品集成测试，还可以使用一些界面测试工具如：SQA Robot, QARun, WinRunner 等。

对于一些基于协议的嵌入式软件，可能需要借助一些消息生成工具、协议模拟仪器、消息跟踪器等。例如可以用于无线协议测试的仪器有：NetHawk 的协议卡、MGTS、3GTS 等。

对于使用 SDL 开发的软件系统来说，使用 TTCN 进行集成测试是非常有利的，Tellogic 公司的 Tau 集成工具中的 Itex 提供这方面的功能。

关于如何选择自己开发测试工具还是购买商用测试工具的分析可以参考笔者即将出版的书籍《软件测试自动化》。

7.8 集成测试应坚持的原则

集成测试是一个灰色地带，要做好集成测试不是一件容易的事情，很多公司在对其开发的软件产品测试过程中，往往忽略了这个过程，究其原因是因为该测试不好把握。另外在很多公司中使用联调（使用调试的手段把模块或子系统一个一个集成起来）来代替集成测试。就如第 1 章所说的，调试与测试有着明显的区别，是不能相互替换的。

集成测试应当针对概要设计尽早开始筹划，为了做好集成测试，需要遵循下列原则，这些原则不是绝对的，仅供参考。

- 所有公共的接口都必须被测试到；
- 关键模块必须进行充分的测试；
- 集成测试应当按一定的层次进行；
- 集成测试的策略选择应当综合考虑质量、成本和进度三者之间的关系；
- 集成测试应当尽早开始，并以概要设计为基础；
- 在模块和接口划分上，测试人员应当和开发人员充分的沟通；
- 当测试计划中的结束标准满足时，集成测试结束；
- 当接口发生修改时，涉及的相关接口都必须进行回归测试；
- 集成测试根据集成测试计划和方案进行，排除测试的随意性；
- 项目管理者保证测试用例经过审核；

- 测试执行结果应当如实的被记录。

7.9 本章小结

集成测试是一个由单元到系统的过渡性测试,由于其位置的特殊性,集成测试往往容易被忽视。对于集成测试的研究业界已经提供了很多的方法,并且很多著作都涉及到了这一方面的内容^{[2][26][126][127][128]}。集成测试策略给出了进行集成测试的一个思路,最常见的集成测试策略有自底向上集成、自顶向下集成、三明治集成、基于集成等。对于面向对象系统使用较多的集成策略有基于线程的集成和基于使用的集成。一般来说,对于一个大的系统,其使用的集成策略往往会综合多种集成策略,策略的选择需要根据其逻辑层次特性和物理分布特性来考虑。

在进行集成分析时需要考虑整个系统的体系结构,包括系统层次关系和依赖关系;需要分析系统的模块,尤其是确定关键模块;需要进行接口分析,划分接口类型,根据不同的接口进行数据分析;需要进行风险分析,分析可能出现的技术风险、人员风险、物料仪器风险、管理风险和市场风险;需要进行可测试分析,以便提前为测试的实现做好准备;常见集成测试故障分析有助于我们进行测试数据的选择和故障预防。

集成测试用例设计类似于单元测试用例设计,可以从以下几个维度考虑:为系统运行起来而设计用例,为正向测试而设计用例,为逆向测试而设计用例,为满足特殊需求而设计用例,为高覆盖设计用例和测试用例补充。

从过程上看集成测试可以分为计划阶段、设计阶段、实现阶段和执行阶段4个阶段。其中计划阶段关注于测试对象范围、工作量、进度、资源、可能存在的风险等因素;设计阶段完成测试分析,包括结构分析、模块分析、接口分析、策略分析、环境分析等等;实现阶段主要完成用例设计,规程设计以及测试脚本;执行阶段完成对测试对象的测试执行工作,并输出测试报告和问题单。

第8章 系统测试

从传统的 V 模型来看,系统测试是产品提交给用户之前进行的最后阶段测试。因此,很多公司把系统测试看成是产品的最后一道防线。系统测试属于黑盒测试范畴,是应用最为广泛的一种测试方法。本章从以下几个方面来介绍有关系统测试的概念:

1. 什么是系统测试?
2. 系统测试的内容有哪些?
3. 一个系统测试的过程是什么?
4. 系统测试总结。

系统测试的内容比较烦杂,涉及面也较广,在本章的写作过程中,综合参考了附录 E 中的一些资料^{[2][5][6][7][10][26][113][148][149][150]},并结合作者多年来从事系统测试的经验进行了总结。

8.1 系统测试概念

系统测试(System Testing)是将已经集成好的软件系统,作为整个基于计算机系统的一个元素,与计算机硬件、外设、某些支持软件、数据和人员等其他系统元素结合在一起,在实际运行(使用)环境下,对计算机系统进行一系列的组装测试和确认测试。

系统测试的目的在于通过与系统的需求定义作比较,发现软件与系统定义不符合或与之矛盾的地方,以验证软件系统的功能和性能等满足其规约所指定的要求。系统测试的测试用例应根据需求分析说明书来设计,并在实际使用环境下来运行。

由于软件只是计算机系统中的一个组成部分,软件开发完成以后,最终还要与系统中其他部分配套运行。系统在投入运行以前各部分需完成组装和确认测试,以保证各组成部分不仅能单独地受到检验,而且在系统各部分协调工作的环境下也能正常工作。这里所说的系统组成部分除软件外,还可能包括计算机硬件及其相关的外围设备、数据及其收集和传输机构、掌握计算机系统运行的人员及其操作等,甚至还可能包括受计算控制的执行机构。显然,系统的确认测试已经完全超出了软件工作的范围。然而,软件在系统中毕竟占有相当重要的位置,软件的质量如何,软件的测试工作进行得是否扎实势必与能否顺利、成功地完成系统测试关系极大。另一方面,系统测试实际上是针对系统中各个组成部分进行的综合性检验。尽管每一个检验有着特定的目标,然而所有的检测工作都要验证系统中每个部分均已得到正确的集成,并能完成指定的功能。

系统测试应该按照测试计划进行,其输入、输出和其他动态运行行为应该与软件规约进行对比。软件系统测试方法很多,主要有功能测试、性能测试、随机测试等。在后面章节将进行详细的描述。

8.2 系统测试方法

系统测试的方法有很多，本章我们将对业界一些比较常见的系统测试方法进行概念性的阐述，关于一些方法的具体应用请参考有关参考文献或笔者即将出版的《软件测试技术研究》和《软件测试实战》。

8.2.1 功能测试

1. 基本概念

功能测试(Functional Testing)是系统测试中最基本的测试，它不管软件内部的实现逻辑，主要根据产品的需求规格说明书和测试需求列表，验证产品的功能实现是否符合产品的需求规格。功能测试主要是为了发现以下几类错误：

- 是否有不正确或遗漏了的功能？
- 功能实现是否满足用户需求和系统设计的隐藏需求？
- 能否正确地接受输入？能否正确地输出结果？

功能测试要求测试设计者对产品的规格说明、需求文档、产品业务功能都非常熟悉，同时对测试用例的设计方法也有一定掌握，才能设计出好的测试方案和测试用例，高效地进行功能测试。

2. 分析方法

在进行功能测试时，首先需要对需求规格进行分析，因为这是功能测试的基本输入。对需求规格的分析可以分为以下几个步骤：

- 对每个明确的功能需求进行标号(对于在需求规格文档中已经有标号的可以直接引用)；
- 对每个可能隐含的功能需求进行标号；
- 对于可能出现的功能异常进行分类分析，并标号；
- 对于前面3个步骤获得的功能需求进行分级；由于我们不可能测试任何东西，因此可以根据风险来决定对每个功能投入多少关注。一般来说，可以把功能划分为关键功能和非关键功能。其中关键功能是指那些对用户来说必不可少的功能，这类功能的丧失将导致用户拒绝产品。例如对于 Microsoft Word 来说，向文档中添加文本是一个关键功能。有时虽然每个单个的功能不是很关键，但是当这些功能综合成一个整体就成为关键功能了，例如 Word 的画图工具条。而非关键功能主要是那些对产品可用性有贡献的功能，这类功能的丧失用户可能会不满意，但不会导致拒绝产品；
- 对每个功能进行测试分析，分析其是否可测、如何测试、可能的输入、可能的输出等；
- 脚本化和自动化。

3. 用例设计

功能测试常用的用例设计方法有：

- 规范导出法；
- 等价类划分；
- 边界值分析；
- 因果图；
- 判定表；
- 正交实验设计；
- 基于风险的测试；
- 错误猜测法。

8.2.2 协议一致性测试

1. 基本概念

在分布式系统中，计算功能，如处理能力、信息存储和人机交互是分布于不同的计算机系统之中的，因而需要这些系统之间能够进行大量的信息交换。为了使各计算机系统能够成功地进行通信，必须遵守一组规则。协议(Protocol)规定了一个计算机系统在和其他计算机系统通信时应遵守的规则集合。为了使得来自于不同厂家的系统能够成功地进行通信，必须有国际化的标准协议。这种需求导致了 OSI(开放系统互联：Open Systems Interconnection)参考模型的制订。但是，描述一组协议并对其标准化并不能保证成功地通信。这是因为协议标准目前基本上是使用自然语言描述的，实现者对于协议的不同理解会导致不同的协议实现，有时甚至是错误的实现。因此，我们需要一种有效的方法对协议实现进行判别，这就是“协议测试”(Protocol Testing)。协议测试包含 4 种测试^[51]：

- 协议一致性测试(Protocol Conformance Testing) 检测实现的系统与标准协议符合的程度；
- 协议性能测试(Protocol Performance Testing) 检测协议实体或者系统的性能指标(数据传输率、连接时间、执行速度等)；
- 协议互操作性测试(Protocol Interoperability Testing) 检测同一协议在不同实现版本之间的互通能力和互连操作能力；
- 协议健壮性测试(Protocol Robustness Testing) 检测协议实体或系统在各种恶劣环境下运行的能力(信道被切断、掉电、注入干扰信息等)。

在这 4 种测试中，只有协议一致性测试能够给出理论框架和方法论。在 20 世纪 90 年代，国际标准组织专门制订了一套国际标准——ISO9646，该标准描述了一个用于测试产品与相关协议标准一致性的通用方法，该方法包含了 7 个部分：X.290 协议描述了基本概念；X.291 协议描述了抽象测试集技术规范；X.292 协议描述了树表组合表示方法 TTCN (the Tree and Tabular Combined Notation)；X.293 协议描述了测试实现；X.294 协议描述了对测试实验室和客户在一致性评价方面的要求；X.295 协议描述了协议轮廓测试；X.296 协议描述了 ICSs(Implementation Conformance Statements)一致性实现语句。

一个系统的一致性可以在两个级别上进行表达,一个是与每个单独的 ITU-T(国际电信联盟的电信标准: the telecommunications standardization sector of International Telecommunications Union)建议或国际标准的一致性,二是与一组相互有关的 ITU-T 建议或国际标准的一致性。一个一致性的系统要能够满足动态和静态一致性的要求,并且符合 PICS(协议实现一致性语句: Protocol Implementation Conformance Statements)的要求和能力。协议一致性测试的主要目的是提高不同系统之间能够互通的概率。但这并不是充分条件,即使两个实现都与同一个协议规范一致,它们也有可能完全不能互通。

2. 分析技术

一致性测试的目标是测试对象与相关协议的一致性。然而,在现实中,我们并不能进行穷尽测试,经济上的考虑会限制人们去做更进一步的测试。因此,可以从下面4个方面来进行协议一致性测试。

- 基本互联测试:该测试提供 ITU 要求符合的基本特征;
- 能力测试:该测试检查 ITU 可观察能力是否符合 PICS 中提出的静态一致性要求和能力;
- 行为测试:提供一种全面的综合测试,即在 IUT(Implementation Under Test)能力之内,由国际标准规定的动态一致性要求的整个范围;
- 定向诊断测试:根据特定的要求,对 IUT 的一致性进行深度探索,以提供一种是非的肯定回答,以及提供与特定一致性有关的诊断信息,这个测试是非标准化的。

协议一致性评价过程是评估某实现或系统与一个或多个 ITU-T 建议或国际标准相一致的所有一致性测试活动的全过程。它分3个阶段:(1)测试准备;(2)测试操作;(3)测试报告的生成。协议一致性评估过程主要有5个步骤。

第一步:IUT 的 PICS 分析,它包括分析该 PICS 与有关标准指定的 SCR(Static Conformance Requirement,静态一致性要求)相一致。

第二步:测试选择,PICS 和 PIXIT(Protocol Implementation Extra Information Statement,协议实现额外声明)被用来从一致性测试集中选取适当的测试例,并使用 PIXIT 提供的信息来量化这些测试例,从 ATS(Abstract Test Suite,抽象测试集)生成可执行测试集,最终产生参数化的可执行测试集。

第三步:测试执行,它由3个方面组成:基本互联测试(可选)、能力测试和行为测试。如果执行基本互联测试。则可在更高级的全面测试之前检查出非一致性情况。能力测试被用来检查 PICS 的有效性和 ITU 应有的性能。行为测试是主要的,它在广泛的测试用例下检查 IUT 的行为,它既包括简单的情况又包括复杂的情况;在与 ITU 的通信中既包含正确的也包含错误的行为,它不能证明一个协议实现在所有通信情形下都是动态一致的,但能证明该实现在所代表的通信实例(测试例)下是动态一致的。

第四步:结果分析,它可以插入测试例中,也可在测试执行步骤结束后考虑。它既是关于每个测试实例的测试效果,也是关于所观察的行为的有效性。

第五步:最终一致性考查,它包括行为测试与已知的 PICS 分析及能力测试结果的综合。从中可以看出这些结果是否与 PICS 表述的能力相一致,从而可获得 ITU 关于标准要

求的一致性结论，这些结果记录于一致性测试报告中。

图 8-1 是协议一致性测试过程的示意图。

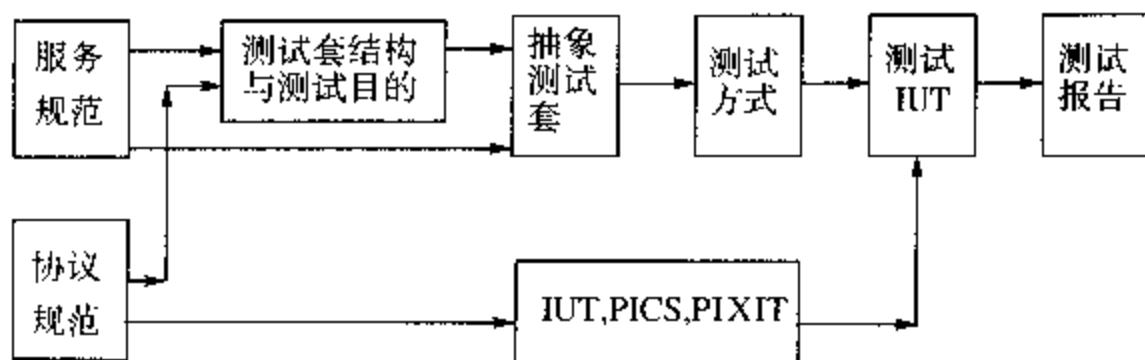


图 8-1 协议一致性测试过程

3. 用例设计

协议测试常用的用例设计方法有：

- 规范导出法；
- 等价类划分；
- 边界值分析。

8.2.3 性能测试

1. 基本概念

在实时系统和嵌入系统中，提供符合功能需求但不符合性能需求的软件是不能被接受的。性能测试(Performance Testing)就是用来测试软件在集成系统中的运行性能的。性能测试可以发生在测试过程的所有步骤中，即使是在单元层，一个单独模块的性能也可以使用白盒测试来进行评估，然而，只有当整个系统的所有成分都集成到一起之后，才能检查一个系统的真正性能。

性能测试的目标是度量系统相对于预定义目标的差距。需要的性能级别针对于实际的性能级别进行比较，并把其中的差距文档化。

一个有用的性能测试是压力测试(Stress Testing)(参考 8.2.4 节内容)，它包括庞大数量的用户和请求来获得操作系统的压力条件。压力测试用于试图耗尽如缓冲区、队列、表、和端口方面的资源限制。这种形式的测试在评价针对拒绝服务方面的危险非常有帮助。

性能测试经常和压力测试一起进行，而且常常需要硬件和软件测试设备，也就是说，在一种苛刻的环境中衡量资源的使用(比如，处理器周期)常常是必要的。外部的测试设备可以监测执行的间歇，当出现情况(如中断)时记录下来。通过对系统的检测，测试者可以发现导致效率降低和系统故障的原因。

性能测试必须要有工具支持，在某些情况下，不得不自己开发专门的接口工具。市面上有一些专门用于 GUI 或 Web 的性能测试工具，具体可以参考笔者即将出版的《软件测试自动化》一书。

2. 分析技术

性能测试是一个混合了黑盒测试和白盒测试的方法。从黑盒测试角度来看,性能分析师不需要知道系统内部的工作原理。根据实际的工作负荷或者基准来比较一个系统版本与另一个系统版本在性能上的改进或降低。从白盒测试观点来看,性能分析师需要知道系统的内部工作原理并且定义特殊的系统资源来进行检查,例如指令、模块和任务等。

一些感兴趣的性能信息包括^[8]:

- CUP 使用情况;
- IO 使用情况;
- 每个指令的 IO 数量;
- 信道使用情况;
- 主要存储内存使用情况;
- 第二存储内存使用情况;
- 每个模块执行时间百分比;
- 一个模块等待 IO 完工的百分比时间;
- 模块使用在主存储上的时间百分比;
- 指令随时间的跟踪路径;
- 控制从一个模块到另一个模块的次数;
- 遇到每一组指令等待的次数;
- 每一组指令页换入和换出的次数;
- 系统反应时间;
- 系统吞吐量,即,每个时间单元的处理数量;
- 所有主要指令的单元执行时间。

基本的性能度量应当首先以没有争议的方式在主要的功能上被执行,例如当一单个任务在正常环境下进行功能执行时的单位质量,这很容易通过秒表来完成。下一套度量应当在系统处于竞争模式下进行,在这种模式下多个任务在进行操作并且排队请求公共资源,例如 CPU、内存、磁盘、信道、网络等。竞争性的系统执行时间和资源使用性能度量通过检视系统来执行以确定潜在无效的区域。

有两种方法收集系统执行时间和资源使用情况。在第一种方法中,系统在其典型的环境中执行用例并使用外部探针、性能监视器或者秒表进行度量。在第二种方法中,探针被插入到代码中,例如,调用一个性能检视程序来收集性能信息。下面是每种方法的讨论,包括测试驱动的讨论,测试驱动是一种支持技术用于收集性能研究的信息。

监视器方法(Monitoring Approach)。这种方法通过确定在一个时间段间隔内根据系统的状态类来检测系统的性能,并且由测试工具或者操作系统的时间设备来控制。在每个时间间隔内,通过采样来显示性能标准的状态。时间间隔越短,采样越精确。

通过监视器收集的统计信息被修正和总结。

探针方法(Probe Approach)。该方法通过插入探针或程序指令到系统程序的许多地方中。例如,为了确定执行一个顺序语句需要的 CPU 时间,第一个探针记录了第一次调用数据收集程序的 CPU 时钟,第二个探针记录了第二次调用数据收集程序的 CPU 时钟。两

个相减就得到了净 CPU 使用时间。可以产生报告显示语句、模块的执行时间。

这些方法的价值是它们被用作性能需求验证工具。然而，正式定义的性能需求必须被描述和设计，这样性能需求可以被跟踪到特定的系统模块。

测试驱动(Test Drivers)。在许多测试用例中，需要测试驱动和测试桩(Test Harnesses)来进行系统性能度量。一个测试驱动提供了执行系统需要的设施，例如，输入。系统需要的输入数据文件被载入，数据文件包含的数据值代表测试的状态以产生记录的数据，并根据期望的结果进行评价。数据通过一种外部格式被产生并提交给系统。

性能测试用例需要被定义，并且需要建立测试脚本。在性能测试被执行之前，性能分析师必须确定目标系统是相对稳定的。否则，许多时间将花在记录和修正缺陷上面，而不是分析性能上面。

下面是用于任何性能研究方面建议的步骤：

- 文档化性能目标，例如，确切的性能度量标准必须被验证；
- 定义测试驱动或者用于驱动系统的输入源；
- 定义要使用的性能方法或者工具；
- 定义性能研究如何被进行，例如，什么是基线，什么是变化的，当可重复的时候如何被验证，如何知道何时研究被完成了？
- 定义报告过程，例如，技术和工具。

3. 用例设计

性能测试常用的用例设计方法有：

- 规范导出法；
- 错误猜测法。

8.2.4 压力测试

1. 基本概念

在较早的软件测试步骤中，白盒测试和黑盒测试技术对正常的程序功能和性能进行了详尽的检查。压力测试(Stress Testing)的目的是对付非正常的情形。在本质上说，进行压力测试的人应该这样问：“我们能够将系统折腾到什么程度而又不会出错？”

压力测试的目的是调查系统在其资源超负荷的情况下的表现。尤其感兴趣的是这些对系统的处理时间有什么影响。这类测试在一种需要反常数量、频率或资源的方式下执行系统。例如，(1)当平均每秒出现1个或2个中断的情形下，应当对每秒出现10个中断的情形来进行特殊的测试；(2)把输入数据的量提高一个数量级来测试输入功能会如何响应；(3)应当执行需要最大的内存或其他资源的测试实例；(4)使用一个虚拟的操作系统中会引起颠簸的测试实例；(5)可能会引起大量的驻留磁盘数据的测试实例。从本质上来说，测试者是想要破坏程序。

压力测试的一个变种是一种被称为是敏感测试(Sensitive Testing)的技术。在某些情况(最常见的是在数学算法中)下，有效数据界限之内的一个很小范围的数据可能会引起极端的甚至是错误的运行，或者引起性能的急剧下降，这种情形和数学函数中的奇点相类

似。敏感测试就是要发现在有效数据输入中可能会引发不稳定或者错误处理的数据组合。

2. 分析技术

压力测试是边界测试。例如，测试最大的活动终端数量，然后加入比需求规定更多的终端。一些负载测试的资源达到了超负荷，这些资源包括：

- 缓冲区；
- 控制器；
- 显示终端；
- 中断处理；
- 内存；
- 网络；
- 打印机；
- 存储设备；
- 事务队列；
- 事务程序；
- 系统用户。

压力测试研究系统在一个短时间内活动处在峰值时的反应。它通常容易同容量测试 (Volume Testing) 混淆，在容量测试中，其目标是检测系统处理大容量数据方面的能力。

压力测试应当在开发过程中尽早进行，因为它通常发现主要的设计缺陷，这些缺陷会影响很多区域。如果压力测试不尽早进行，一些在开发早期可能很明显的细微的缺陷难于被发现。

下面是压力测试建议的一些步骤：

- 进行简单的多任务测试；
- 在简单的压力缺陷被修正后，增加系统的压力直到中断；
- 在每个版本循环中重复进行压力测试。

一些压力测试的例子包括：

- 对于一个固定输入速率的单词处理响应时间，例如每分钟 120 个单词；
- 在一个非常短的时间内引入超负荷的数据容量；
- 改变交互、实时、过程控制方面的负荷；
- 同时引入大量的操作；
- 成千上万的用户在同一时间登录到 Internet 上。

3. 用例设计

压力测试常用的用例设计方法有：

- 规范导出法；
- 边界值分析；
- 错误猜测法。

8.2.5 容量测试

1. 基本概念

容量测试 (Volume Testing) 的目的是使系统承受超额的数据容量来发现它是否能够正确处理。这种测试通常容易同压力测试混淆。压力测试主要是使系统承受速度方面的超额负载, 例如一个短时间之内的吞吐量。容量测试是面向数据的, 并且它的目的是显示系统可以处理目标内确定的数据容量。

2. 分析方法

进行容量测试一般可以通过以下几个步骤来完成:

- (1) 分析系统的外部数据源, 并进行分类;
- (2) 对每类数据源分析可能的容量限制, 对于记录类型数据需要分析记录长度限制、记录中每个域长度限制和记录数量限制;
- (3) 对每个类型数据源, 构造大容量数据对系统进行测试;
- (4) 分析测试结果, 并与期望值比较, 确定目前系统的容量瓶颈;
- (5) 对系统进行优化并重复(1)~(4)步骤, 直到系统达到期望的容量处理能力。

常见的容量测试例子包括:

- 当处理数据敏感操作时进行的相关数据比较;
- 使用编译器编译一个极其庞大的源程序;
- 使用一个链接编辑器编辑一个包含成千上万模块的程序;
- 一个电路模拟器模拟包含成千上万模块的电路;
- 一个操作系统的任务队列被充满;
- 一个测试形式的系统被灌输了大量文档格式;
- 庞大的 E-mail 信息和文件充满了 Internet。

3. 用例设计

容量测试常用的用例设计方法有:

- 规范导出法;
- 边界值分析;
- 错误猜测法。

8.2.6 安全性测试

1. 基本概念

任何管理敏感信息或者能够对个人造成不正当伤害的计算机系统都是不正当的或非法的侵入目标。侵入包括了范围很广的活动: 只是为练习而试图侵入系统的黑客; 为了报复而试图攻破系统的有怨言的雇员; 还有为了得到非法的利益而试图侵入系统的不诚实的

个人。

安全测试(Security Testing)用来验证集成在系统内的保护机制是否能够在实际中保护系统不受到非法的侵入。引用 Beizer 的话来说:“系统的安全当然必须能够经受住正面的攻击——但是它也必须能够经受住侧面的和背后的攻击。”^[26]安全性测试应当被设计以证明资源如何被保护^[8]。

在安全测试过程中,测试者扮演着一个试图攻击系统的个人角色。就是这样!测试者可以尝试通过外部的手段来获取系统的密码,可以使用能够瓦解任何防守的客户软件来攻击系统;可以把系统“制服”,使得别人无法访问;可以有目的地引发系统错误,期望在系统恢复过程中侵入系统;可以通过浏览非保密的数据,从中找到进入系统的钥匙等。只要有足够的时间和资源,好的安全测试就一定能够侵入一个系统。系统设计者的任务就是要把系统设计为想要攻破系统而付出的代价大于攻破系统之后得到的信息价值。

保护测试(Protection Testing)是安全测试中一种常见的测试,主要用于测试系统的信息保护机制,类似资料可以参考附录 E^[153]。

2. 分析方法

设计安全性测试用例的策略需要包含4个方面的分析:资产、危险、暴露出来的行为和安全性控制。在这种方式下,矩阵和检查表是安全性测试用例的建议方法。

资产是一个实体确切的和非确切的资源。评价方法是列出什么应当被保护。这也有助于检查资产的属性,例如数量、价值、使用和特性。两个有用的分析技术是资产的价值和使用分析。资产价值分析确定资产的价值在用户和潜在的攻击者之间是如何地不同。资产使用分析检查非法使用资产的不同方法。

危险是可能引起损失或者伤害的事件。评价方法是列出潜在危险的源头。区别意外、故意和自然的危险以及危险的频率是非常重要的。

暴露出来的行为是可能的损失或者伤害的形式。评价的方法是列出一个危险发生时会对资产发生什么事情。暴露出来的行为包括侵害、错误的决定和欺骗。暴露出来的行为分析关注于该行为影响最大的区域。

安全性功能或者控制是度量针对于损失或者伤害的保护的程度。评价的方法是列出安全性功能和任务并且关注于在特定系统功能或者过程中包含的控制。安全性功能评价针对人为错误和偶然的系统误操作。一些功能性的安全性问题包括:

- 控制特性是否工作正确?
- 无效的或者不可能的参数是否被检测并且适当地处理?
- 无效的或者超出范围的指令是否被检测并且适当地处理?
- 错误和文件访问是否适当地被记录?
- 是否有变更安全性表格的过程?
- 系统配置数据是否能正确保存,系统故障时是否能恢复?
- 系统配置数据能否导出,在其他机器上进行备份?
- 系统配置数据能否导入,导入后能否正常使用?
- 系统配置数据保存时是否加密?
- 没有口令是否可以登录到系统中?

- 有效的口令是否被接受,无效的口令是否被拒绝?
- 系统对多次无效口令是否有适当的反应?
- 系统初始的权限功能是否正确?
- 各级用户权限划分是否合理?
- 用户的生命期是否有限制?
- 低级别的用户是否可以操作高级别用户命令?
- 高级别用户是否可以操作低级别用户命令?
- 用户是否会超时退出,超时的时间是否设置合理,用户数据是否会丢失?
- 登录用户修改其他用户的参数是否立即生效?
- 系统在最大用户数量时是否操作正常?
- 对于远端操作是否有安全方面的特性?
- 防火墙是否能被激活和取消激活?
- 防火墙功能激活后是否会引发其他问题?

评价安全机制的性能与安全性功能本身一样重要。一些问题就集中在安全性的性能上,包括:

- 有效性(Availability)

执行严格的安全性功能所占有的时间比例?安全性控制一般需要比系统其他部分更高的有效性。

- 生存性(Survivability)

系统在抵制主要错误或者自然灾害方面的能力如何?这包括错误期间紧急操作的支持、随后的备份操作和恢复到正常操作的功能。

- 精确性(Accuracy)

安全性控制精确度如何?精确性围绕错误的数量、频率和严重性。

- 反应时间(Response Time)

反应时间是否可接受?慢的反应时间可能导致用户绕过安全控制。当安全表格动态修改时,反应时间还对控制管理很关键。

- 吞吐量(Throughput)

安全性控制是否支持需要的使用吞吐量?吞吐量包含用户和服务请求的峰值和平均值。

3. 用例设计

安全性测试常用的用例设计方法有:

- 规范导出法;
- 边界值分析;
- 错误猜测法;
- 基于风险的测试;
- 故障插入技术。

8.2.7 恢复性测试

1. 基本概念

很多基于计算机的系统必须在一定的时间内从错误中恢复过来，然后继续运行。在某些情况下，系统必须是可以容错的，也就是说，运行过程中的错误不能使得整个系统的功能都停止。在其他情况下，一个系统错误必须在一个特定的时间段内改正，否则就会产生严重的经济损失。

恢复测试(Recovery Testing)的目标是验证系统从软件或者硬件失败中恢复的能力。这个测试验证系统在处理过程中处理中断和回到特殊点的偶然特性。恢复测试采取各种人工干预方式使软件出错，而不能正常工作，进而检验系统的恢复能力。如果系统本身能够自动地进行恢复，则应检验：重新初始化、检验点设置机构、数据恢复以及重新启动是否正确。如果这一恢复需要人为干预，则应考虑平均修复时间是否在限定的范围以内

2. 分析技术

在设计恢复性测试用例时，需要考虑下面这些关键问题：

- 是否存在潜在的灾难和已确认的系统失败，以及它们的损失？消防训练式的头脑风暴法是定义灾难场景方面的一个有效方法；
- 保护和恢复过程是否为错误提供了足够的反应？计划过程应当使用技术评审来进行测试，评审人员包括主要事件专家和系统用户；
- 当真正需要时，恢复过程是否能够正确工作？模拟的灾难需要和实际的系统一起被创建以验证恢复过程。这应当和用户、支持组织、供应商交涉在一起。

一些恢复性测试的例子包括：

- 完全的恢复在日常维护期间或者错误恢复期间备份的文件；
- 恢复部分文件以回到上个检查点；
- 恢复程序的执行；
- 对选择的文件和数据进行恢复；
- 当供电出现问题时的恢复；
- 手工恢复过程的验证；
- 通过切换到一个并行系统来进行恢复；
- 恢复操作引起系统性能的降级；
- 恢复期间的安全性过程；
- 恢复处理日志方面的能力。

3. 用例设计

恢复性测试常用的用例设计方法有：

- 规范导出法；
- 错误猜测法；
- 基于故障的测试；

- 基于风险的测试。

8.2.8 备份测试

1. 基本概念

备份测试(Backup Testing)是恢复性测试的一个补充,并且应当是恢复性测试的一个部分。备份测试的目的是验证系统在软件或者硬件失败的事件中备份它数据的能力。

2. 分析技术

备份测试需要从以下几个角度来进行设计:

- 备份文件,并且比较备份文件与最初的文件的区别;
- 存储文件和数据;
- 完全的系统备份步骤;
- 检查点备份;
- 备份引起系统性能的降级;
- 手工操作过程备份的有效性;
- 系统备份“触发器”的检测;
- 备份期间的安全性过程;
- 备份过程期间的维护处理日志。

3. 用例设计

可参考恢复性测试的相关内容。

8.2.9 GUI 测试

1. 基本概念

软件业的迅速发展使得软件产品越来越深入到每个人的生活当中。因此对于一个不懂电脑的人员如何能够快速接受并学会使用软件产品就变得非常关键。图形化用户接口 GUI(Graphic User Interface)已经越来越成为用户喜欢的一种人机接口界面^[157],因此 GUI 的好坏将直接影响用户使用软件时的效率及心情,也直接影响用户对所使用的系统的印象。为了让软件能够更好地服务于用户,进行 GUI 测试就变得非常重要了。GUI 测试与用户友好性测试和可操作性测试有重复,但 GUI 测试更关注的是对图形界面的测试。

什么是 GUI? 一个 GUI 是一个分层的图形化的软件前端,它从一个固定的事件集中接受用户产生的和系统产生的事件,并且产生确定的图形输出。一个 GUI 包含许多图形对象(例如:菜单、按钮、列表框、边界框等),每个图形对象有一个固定的属性集合。在 GUI 执行的任何时刻,这些属性有着离散的值,这些值的集合组成了 GUI 的状态。^[160]

上述 GUI 定义并不是严格的,它只代表了某一类的 GUI。这个定义需要被扩展到能够包含基于网络用户接口或视频接口领域。在此就不详细说明了。

GUI 测试主要包括两方面的内容,一方面是界面实现与界面设计的吻合情况;另一方面是确认界面处理的正确性。界面设计与实现是否吻合,主要指界面的外形是否与设计内容一致;界面处理的正确性也就是当界面元素被赋予各种值时,系统处理是否符合设计以及是否没有异常。例如,当我们选择“打开文档”菜单,系统应当弹出一个打开文档的对话框,而不是弹出一个保存文档的对话框或别的对话框。

2. 分析技术

GUI 测试是困难的,这主要因为有几个原因^{[159][161]}:

- 一个 GUI 的可能接口空间是非常庞大的。每个 GUI 活动序列可能会导致系统处在不同的状态上。一般来说,GUI 活动的结果在系统不同的状态上是不同的。这样就必须在一个庞大状态集上进行 GUI 测试^[162],这个工作一般难以完成,即使借助自动化测试;
- GUI 的事件驱动特性,由于用户可能会单击屏幕上任何一个像素,因此对 GUI 来说可能会产生非常非常多的用户输入。模拟这类输入是困难的;
- GUI 测试的覆盖率不同于传统的结构化覆盖率,理论上不够成熟,且没有合适的自动化工具^[160];
- 界面的美学具有很大的主观性。界面元素的默认大小,元素间的组合及排列次序,界面元素的位置,界面颜色等,这些对不同的用户来说可能会有不同的结果,因此如何才能代表大部分客户的意见也是界面测试的一个难点。尤其测试人员之间在这些方面的不同意见可能会导致开发人员的抵制;
- 糟糕的设计使得界面与功能混杂在一起,这使得界面的修改会导致更多的错误,同时也增加了测试的难度和工作量。

为了更好地进行 GUI 测试,提倡界面与功能的设计分离。一般可以把 GUI 系统分为 3 个层次:界面层、界面与功能的接口层、功能层。因此 GUI 测试可以把重点关注在界面层和界面与功能接口层上。

体现越早测试越好的原则,GUI 测试也应当尽早进行。原型法是一个获取需求,并确定界面的很好的方法,这个方法在项目需求阶段就可以使用,伴随着这个方法,GUI 测试就可以开始进行了。此时的测试可以采用场景测试方法,由测试人员扮演场景中的角色,模拟各种可能的操作以及可能的操作序列,并且由用户来判断是否合理,是否有功能的遗漏。一旦原型基本确定之后,就可以开始着手编写 GUI 测试用例,并且借助 GUI 自动测试工具(在 GUI 测试中,如果不借助自动化测试工具,测试工作将是非常枯燥乏味的,但是如果界面不能尽早确定,因界面的经常变动而导致自动化脚本的维护工作量也是难以接受的。一般业界常用的 GUI 自动化测试工具有 QARun, WinRunner, QARobot, Visual Test 等,具体可以参考 <http://www.cigital.com/marick/faqs/t-gui.htm>)^{[163][164]}。

在设计 GUI 测试用例时,可以从以下几个步骤进行思考。

(1) 划分界面元素,并根据界面的复杂性进行分层

一般可以把界面分为 3 个层次。第一个层次是界面原子,界面原子是指不可再分的界面组成元素,例如:一个菜单项、一个按钮、一个列表框、一个编辑框、工具栏中的一个图标、一个快捷键、一个静态文本等。在这些界面原子中,有的是用于接受输入的;有的

是用于输出显示的；有的是显性的(在界面上可以直接观察到)；有的是隐性的(在界面上无法直接观察到，只能通过某些操作来激活)；有的是在系统的整个生命周期中一直存在的；有的是在系统生命周期过程中动态产生，并会动态消失的；有的是动态的；有的是静态的。

第二个层次是界面组合元素，这些界面元素是由多个具有相同属性的界面原子或者彼此协助的一组界面原子组合而形成的一类界面元素。例如：工具栏、组合框、表格、菜单栏等。

第三个层次是一个完整的窗口。一个完整的窗口是由一系列界面组合元素组成的能够完成一个完整的输入输出功能的界面属性组合，并且它具有自己的视图(View)。例如：一个对话框、一个单文档窗口、多文档系统的父窗口、多文档系统的一个子窗口等。

(2) 在不同的界面层次确定不同的测试策略

一般在界面原子层，主要考虑该界面原子的显示属性、触发机制、功能行为、可能的状态集等内容。对于界面原子可能接受的输入可以从等价类划分，边界值分析等角度考虑，触发机制可以从规范导出的方法分析，功能行为可以使用因果图或判定表，可能的状态集可以使用错误猜测法或基于错误的测试方法等。

对于界面组合元素，主要考虑界面原子组合顺序、排列组合、整体外观、组合后功能行为的多个角度进行测试。

对于一个完整的窗口，主要考虑窗口的整体外观、窗口元素的排列组合、窗口属性值、窗口可能的各种组合行为(或可能的操作路径)等。

(3) 进行测试数据分析，提取测试用例

对于界面元素的外观，可以从以下几个角度获取测试数据：

- 界面元素大小；
- 界面元素形状；
- 界面元素的色彩、对比度、明亮度；
- 界面元素包含的文字属性(如：字体、排序方式、大小等)。

对于界面元素的布局，可以从以下几个角度获取测试数据：

- 各界面元素位置；
- 各界面元素的对齐方式；
- 各界面元素间间隔；
- Tab 顺序；
- 各界面元素间色彩搭配。

对于界面元素的行为，可以从以下几个角度获取测试数据：

- 回显功能；
- 输入限制和输入检查；
- 输入提醒；
- 联机帮助；
- 默认值；
- 激活或取消激活；
- 焦点状态；

- 功能键或快捷键;
 - 操作路径;
 - 行为回退(Undo)。
- (4) 使用自动测试工具进行脚本化工作

3. 用例设计

GUI 测试常用的用例设计方法有:

- 规范导出;
- 等价类划分;
- 边界值分析;
- 因果图;
- 判定表;
- 错误猜测法。

8.2.10 健壮性测试

1. 基本概念

健壮性测试(Robustness Testing)有时也叫容错性测试(Fault Tolerance Testing)。主要用于测试系统在出现故障时,是否能够自动恢复或者忽略故障继续运行。为了创建一个健壮的系统,必须小心地设计和实现你的系统,尤其是在系统的异常处理方面。即,一个健壮的系统是设计出来的而不是测试出来的。

对于一般的软件企业来讲,成本、时间和人员约束经常限制软件测试关注于重要的功能正确性领域,而往往忽略或仅分配少量的资源用于确定系统在异常处理方面的健壮性^[168]。这个矛盾在随着软件应用的日益普遍方面体现得异常突出。因此一个好的软件系统必须经过健壮性测试之后才能最终交付给用户。

2. 分析技术

健壮性测试最一般的方法是软件故障插入测试(Software Fault Insertion Testing, SFIT)(这方面涉及到的一些概念可以参考本书第4章和第5章的相关内容)。该技术模拟在程序代码的特定位出现故障情况并且观察系统的行为。在许多系统中,一个程序的路径可能非常复杂,并且错误出现的触发情况很少出现,在此情况下容错行为的检测就变得非常困难了^[169]。

在软件故障插入测试技术中,需要关注3个方面:目标系统、故障类型和插入故障的方法^[170]。

目标系统包括从实时的分布式依赖系统到大型的操作系统等各类系统^[171]。目标系统的不同会影响故障类型和插入方法。

插入到一个系统中的故障类型变化很大。一般一个计算机的故障可以从它们的原因、严重性和恢复成本这三个维度来进行分类。这方面应用得比较成熟的技术是故障模式。故障模式方面研究的重要性已经越来越被大家所理解,并且出现了很多这方面的研究成果。

表 8-1 给出了业界在这一领域的一些研究成果^[169]。

表 8-1 业界在故障和失效方面的研究成果

机构名称	被测系统	研究内容	研究成果
Bellcore	在 Bell 经营的公司中的存储程序控制交换系统 ^[172]	1. 这些系统的平均当机时间 2. 当机频率	在 2.5 年期间, 平均每年的当机时间是 3.5 分钟 原因: 恢复软件占 30% 硬件占 25% 过程占 42% 其他占 2%
CMU	在 Unix OS 上的矩阵乘积排序程序 ^{[173][174]}	1. 目标程序的错误检测能力有多好 2. 是否存在一个小规模的失效表示集	总结了五种失效模式: 崩溃 (Crash) 中止 (Abort) 延迟 (Delay) 挂起 (Hang) 不正确反馈 (Incorrect Answer)
HP	HP 软件产品 ^{[175][176]}	在 HP 软件产品中软件故障的分布是什么	故障分布情况: 模块接口故障占 56.9% 模块不正确的行为占 6.1% 模块内部故障 9.8% 模块功能故障 27.2%
IBM	IBM 项目 ^{[177][178]}	在开发分析过程中是否存在一种机制来确认、检测和控制缺陷	正交缺陷分析 (ODA: Orthogonal Defect Analysis) 通过获取影响信息, 提供过程内反馈给开发人员
	MVS 操作系统	软件系统是如何影响系统的可用性的	存储覆盖缺陷比一般的缺陷有着更大的危害
Tandem	GUARDIAN90 操作系统 ^[179]	是否能够通过事件日志来分析和确认错误/失效	开发、实现和自动化了一个分析的方法 在报告中的域失效中, 72% 还会再发生
	客户场所的 Tandem 系统	软件是否更应为故障负责	软件是故障的主要根源 (在过去 5 年中, 软件引起的失效从 33% 上升到 62%, 硬件引起的故障从 50% 下降到 10%)

插入故障的方法可以分为两类: 状态插入 (State Injection) 和代码插入 (Code Injection)。状态插入可以通过改变一个运行系统的状态或行为来实现。更特殊的是, 插入一个系统级的错误作为一个错误的状态。实现状态插入有多种方法, 其中包括以下几种。

- 基于进程的 (Process Based): 插入的实现通过一个高优先级的进程来修改计算的状态^[180]。这个方法经常依赖于底层的操作系统;

- 基于调试的(Debugger Based): 使用一个调试器(例如: dbx, gdb), 错误可以通过调试器一些内置功能的组合(例如: 断点, 跳转)来把错误注入到一个运行的系统中;
- 基于消息的(Message Based): 对于在两个组件中面向消息的通信, 使用基于消息的工具, 通过破坏消息序列来创建错误的状态。这种故障类型在 AAS 的测试项目中研究过^[181]。

关于代码插入的有关技术可以参考本书第4章内容。健壮性测试一般需要借助工具来实现, 一些用于软件故障插入测试的工具可以参考表8-2。

表8-2 故障插入工具一览表

	级别	故障类型	故障持续时间	插入方法	目标系统	贡献	不足
MESSA-LINE ^{[190][191]}	硬件	IC PIN	永久	要硬件支持	铁路控制系统	验证工具, 在输入域和故障分布方面比较有见解	级别低
FERRARI ^[189]	硬件和内存/寄存器	地址 程序 计数器	短暂的 永久的	要硬件支持	实时系统 快速排序 矩阵乘积	实时验证工具	级别低
FI-AT ^{[187][188]}	内存/寄存器	内存位	永久	软件实现	复式系统 快速排序 矩阵乘积	在软件级别上的自动故障注入器的设计	级别低
SFI ^{[185][186]}	内存/寄存器	内存故障 通信失效 处理器失败	短暂的 间歇的 永久的	使用高优先级进程	分布式系统	注入使用失效或有故障的行为	级别低
FINE ^[184]	内存/寄存器和软件模块	总线、CPU、内存分配	永久的	具有核心特权的服务器进程	Unix 操作系统	验证工具	级别低
TAMER ^{[154][168][179]}	软件模块	接口错误	永久的	在功能接口上的软件	Unix 应用数据库系统	测试充分性容错	难以度量分布式系统必须为每个测试提供失败模式
PSN ^[183]	软件模块	硬件错误	永久的	软件	实时系统	凸现易错区域, 度量容错能力	在时序方面的有效性不明可接受行为的概念模糊
ORCHESTRA ^[182]	网络通信协议栈	时序错误	永久的	软件(操作消息)	协议测试	验证工具 时序容错	只对协议验证有效补偿性时序特性似乎依赖于平台

在设计一个充分的健壮性测试思路可以从下面几个角度考虑：

- 基于错误的策略
- (1) 确认所有可能的错误源
- (2) 为每一类错误开发错误插入技术

- 基于覆盖率的策略

根据下面几个类型度量覆盖率：

- (1) 接口覆盖的数量
- (2) 故障位置覆盖的数量
- (3) 例外覆盖的数量

- 基于失效的策略

这有助于回答下列问题：

- (1) 用例设计故障是否被处理了
- (2) 例外是否被处理了
- (3) 一个组件中的失效是否影响另一个组件

3. 用例设计

- 故障插入测试；
- 变体测试；
- 错误猜测法。

8.2.11 兼容性测试

1. 基本概念

兼容性测试(Compatibility Testing)的目的是测试应用对其他应用或者系统的兼容性。这种测试经常被忽略，并且这方面的错误通常很微妙且难以发现。一个例子是当系统在受控的测试实验室中工作得很好，但是当它和其他应用一起存在时就不能工作了。兼容性的一个例子是当两个系统同时共享相同的数据或者数据文件或者内存时，系统可能满足系统需求但是不能在一个共享环境中工作并且可能和其他系统交互。

2. 分析技术

进行兼容性测试时，可以按照下面的步骤进行：

- (1) 更新兼容性目标以记录应用是如何被真正开发的，实际的环境是如何运作的。更改目标以适用于共存系统或配置资源中的任何修改；
- (2) 更新兼容性测试用例以保证它们是全面的。保证其他系统中可能影响目标系统的测试用例是全面的。保证最大覆盖一个系统可能影响另一个系统的实例；
- (3) 执行兼容性测试并且仔细检视结果以保证期望的结果。使用基线的方法，基线是额外的目标系统进入到共享环境之前系统的操作特性。基线需要精确和完整，而不仅仅是功能性的，但是运行性能需要保证在各种系统共存时不会降级；
- (4) 文档化兼容性测试的结果并且记录在目标系统和其他共存系统中的任何异常；

(5) 在缺陷修正后,需要重新进行兼容性测试。

一般来说,考虑兼容性测试时需要注意以下问题:

- 当前系统可能运行在哪些不同的操作系统环境下?
- 当前系统可能与哪些不同类型的数据库进行数据交换?
- 当前系统可能在哪些不同的硬件配置的环境上?
- 当前系统可能需要与哪些软件系统协同工作?这些软件系统可能的版本有哪些?
- 上面的这些情况是否需要综合测试?如不同硬件环境下的不同操作系统环境等。

3. 用例设计

- 规范导出法;
- 错误猜测法。

8.2.12 可用性测试

1. 基本概念

可用性测试(Usability Testing)和可操作性测试(Operate Testing)有很大相似性,它们都是为了检测用户在理解和使用系统方面到底有多好。这包括系统功能、系统发布、帮助文本和过程,以保证用户能够舒适地和系统交互。在实际测试时,往往把这两者放到一起进行考虑,很少会去严格区别这两者之间的关系。

可用性测试应当在开发期间尽可能早地进行并且应当被设计到系统中去。被延迟了的可用性测试是不可能的,因为系统已经被锁住,通常需要对系统进行大的重设计以修正严重的可用性错误。这可能在经济上是不适合的。

2. 分析技术

一些测试人员应当关注的可用性问题包括:

- 过分复杂的功能或者指令;
- 困难的安装过程;
- 错误信息过于简单;例如“系统错误”;
- 语法难于理解和使用;
- 非标准的 GUI 接口;
- 用户被迫去记住太多的信息;
- 难以登录;
- 帮助文本上下文不敏感或者不够详细;
- 和其他系统之间的连接太弱;
- 默认不够清晰;
- 接口太简单或者太复杂;
- 语法、格式和定义不一致;
- 没有给用户提供所有输入的清晰的知识。

3. 用例设计

- 规范导出法；
- 错误猜测法。

8.2.13 可安装性测试

1. 基本概念

理想情况下，一个软件的安装程序应当平滑地集成用户的新软件到已有的系统中去，就像一个客人被介绍到一个聚会中去一样，彼此交换适当的问候。一些对话框提供简单的、容易理解的安装选项和支持信息，并且完成安装过程。然而，在某些糟糕的情况下，安装程序可能会做错误的事情，使新的程序无法工作，已有的功能受到影响，甚至安装过程严重损坏用户系统。设想一下，作为一个用户，如果因为安装一个软件导致操作系统崩溃而不得不重新安装系统，那将是多么可怕的事情。同样，如果你是软件开发人员，并且花费了很大的精力设计和验证了一个质量良好的软件，就因为安装问题导致用户拒绝，这将是一件多么令人懊恼和痛苦的事情。

可安装性测试(Installation Testing)的目的就是要验证成功安装系统的能力。安装系统通常是开发人员的最后一个活动，并且通常在开发期间不太受关注。然而，它是客户使用新系统时执行的第一个操作。因此，清晰并且简单的安装过程是系统文档中最重要的部分。

2. 分析技术

在考虑可安装性测试时，需要考虑是否需要包含重新安装过程，以便能够倒退安装过程并且确认先前环境条件。同样，安装过程需要记录用于可以如何调整系统选项并且从先前的一个版本上进行升级。

然而，安装测试，尤其是手工进行安装测试，这可能是非常令人烦恼的一件事情。因为它经常需要花费很长的时间来运行一次安装测试用例，尤其对于一个巨大的软件包。如果要考虑你希望覆盖的各种机器配置，你将会发现这将是一场梦魇。因此自动化安装测试就变得非常关键，Christopher Agruss 从以下几个方面来考虑进行自动化测试^[155]：

- 确认安装程序自动化测试的内置层次。绝大多数安装程序都使用脚本来确定如何把文件写入硬盘，配置你的系统并且完成其他一些必须的修改。在此之前，许多安装程序还进行了某些类型的检测以验证被使用的机器与新软件兼容，有足够的空间等。从这一点看，一个软件安装程序在某种程度上说已经是一个自动测试程序了。因此，我们可以通过安装程序使用的脚本来设计自动化测试用例。这些用例在安装的过程中被自动执行。
- 控制机器的基本状态。对任何安装测试来说，首要的基本步骤是建立一个机器状态基线。磁盘映像程序被广泛地用于把机器硬盘恢复到一个基本状态，如：GHOST，IC3 等。你可能需要一组机器的基本状态映像，这依赖于你想要测试多少配置。一个方法是对于每个操作系统和硬盘文件格式的每个组合保留一个

映像。

- 使用一个测试工具来驱动安装程序。使用企业级用户接口测试工具有许多优势。例如它提供一个恢复系统，一旦进入错误状态时，工具会自动恢复系统。此外这些工具已经有大量的测试库，可以帮助你验证结果。
- 使用流程图来设计你的自动测试。安装测试流程设计的一个好的开端是设计一个安装行为的简单流程图。如图8-2所示。图8-3与图8-4是一个扩展：这些例子相对都比较简单，在实际测试当中，可能还需要考虑更多的情况。

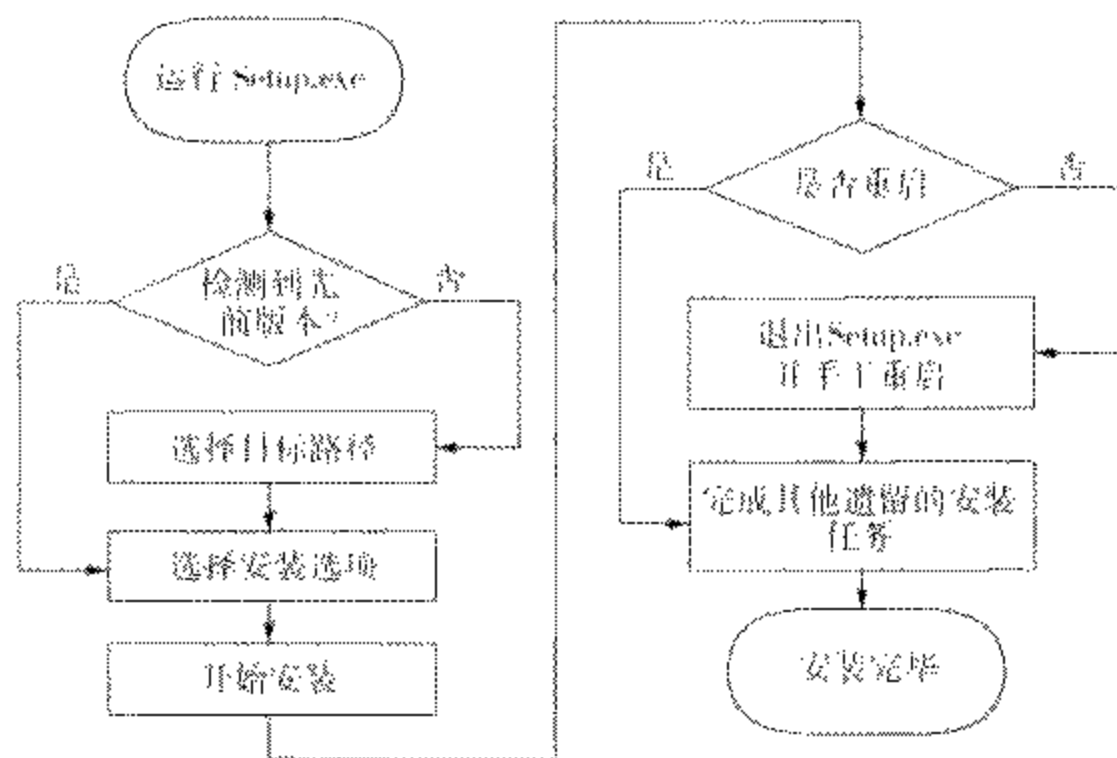


图 8-2 一般安装程序流程图

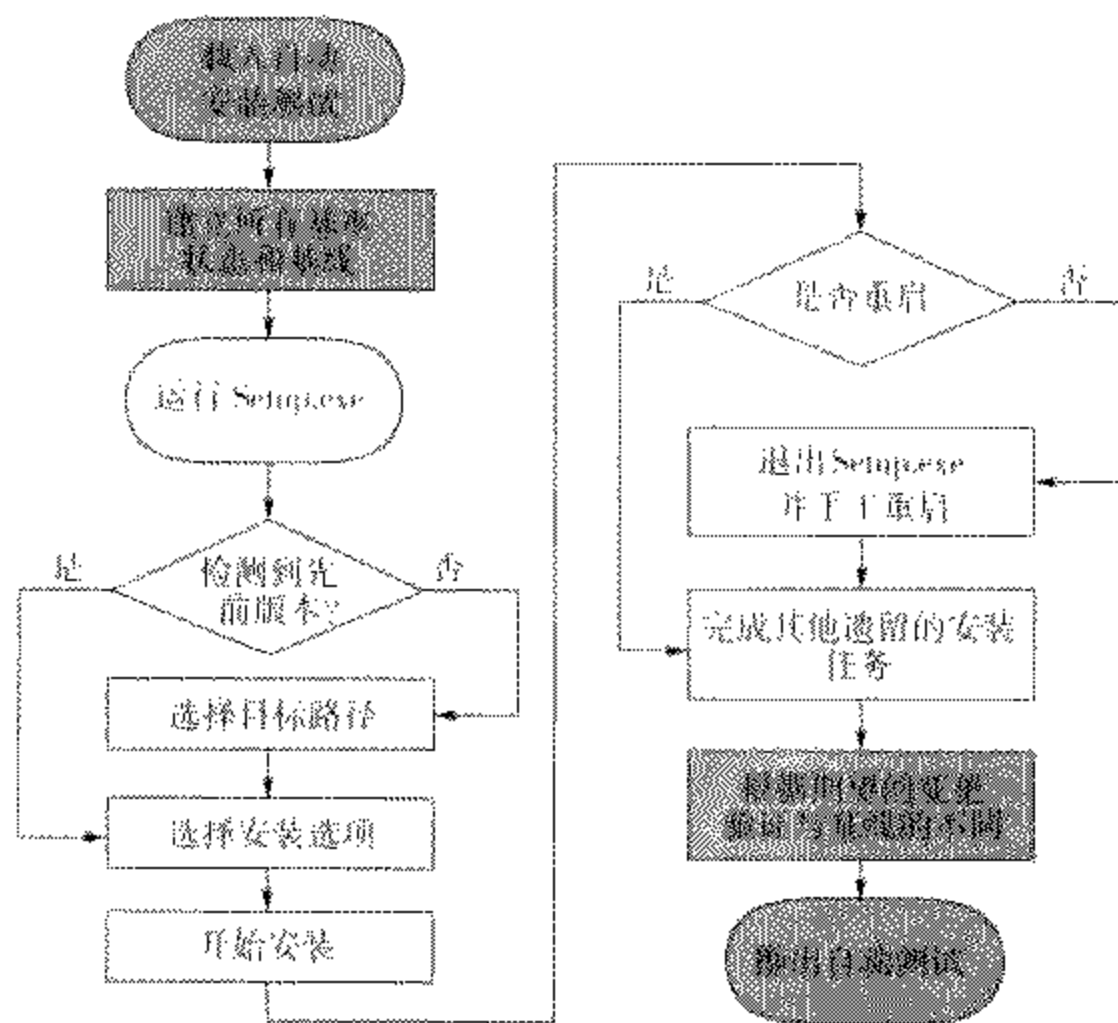


图 8-3 对安装程序使用自动测试的流程图

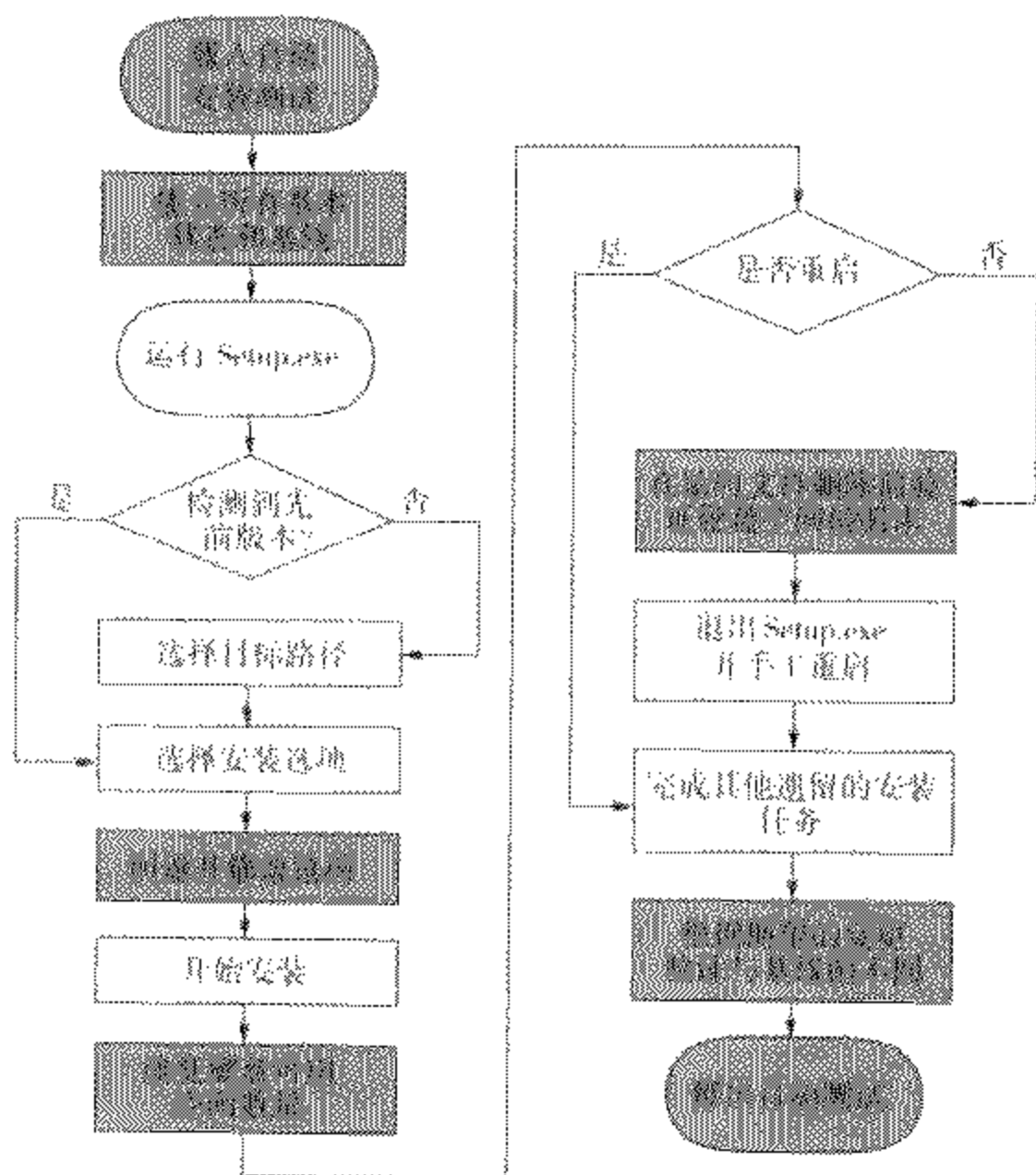


图 8-4 使用自动测试测试安装空间流程图

- 使用多台机器来运行安装测试。有许多方法可以为你的安装测试建立一个分布式测试环境，建议使用主从配置——即一台主机驱动测试在一台或多台从机上执行。可以使用 Rational 的测试管理工具 TestManager。
- 自动验证安装测试的结果。主要是验证安装程序是否把期望的文件安装到期望的位置上。这个问题可以通过创建一个数据驱动测试来解决，这个测试通过读取文件列表来与一个外部表进行比较。
- 实际安装需要硬盘空间的验证。这是安装程序的一个特点，可以直接在安装程序脚本中进行，有些测试工具也提供一些库函数用来报告当前的可用空间，这使得这些计算变得相当简单。
- 自动化测试卸载程序。一些流行的安装程序的一个基本问题是它们只能卸载上一次安装操作的软件，即使这次安装操作只是增加了一些额外的文件到一个更大的已存在的安装操作中。

测试人员在考虑可安装性测试时，可以考虑下面这些关键问题：

- 谁安装系统？例如，假设的技术能力是什么？
- 安装过程是否被完整地文档化，并且有详细且明确的安装步骤？
- 安装过程假定工作在哪个环境中？例如，平台，软件，硬件、网络、版本？

- 安装是否修改用户当前的环境设置？例如，config.sys 等。
- 安装人员如何知道系统已经被正确安装了？例如，是否有一个适当的安装测试过程？

3. 用例设计

- 规范导出法；
- 错误猜测法。

8.2.14 文档测试

1. 基本概念

这里的文档测试不同于评审和检视工作，主要是针对系统提交给用户的文档的验证。文档测试(Documentation Testing)的目标是验证用户文档是正确的并且保证操作手册的过程能够正确工作。文档测试有一些优点，包括改进系统的可用性、可靠性、可维护性和安装性。在这些例子中，测试文档有助于发现系统中的不足并且/或者使得系统更可用。文档测试还减少客户支持成本，例如客户可以知道文档的问题，而不需要叫他们当面提供支持。

2. 分析技术

在文档测试时，测试人员假定自己是用户，按照文档中的说明进行操作。在进行文档测试的时候，可以考虑以下几个方面：

- 使用文档作为许多测试用例的一个源头；
- 确切地按照文档所描述的方法使用系统；
- 测试每个提示和建议；
- 把缺陷并入缺陷跟踪库；
- 测试每个在线帮助超链接；
- 测试每条语句，不要想当然；
- 表现得像一个技术编辑而不是一个被动的评审者；
- 首先对整个文档进行一般的评审，然后进行一个详细的评审；
- 检查所有的错误信息；
- 测试文档中提供的每个样例；
- 保证所有索引的入口有文档文本；
- 保证文档覆盖所有关键用户功能；
- 保证阅读类型不是太技术化；
- 寻找相对比较弱的区域，这些区域需要更多的解释。

3. 用例设计

规范导出法。

8.2.15 在线帮助测试

1. 基本概念

用户在使用系统时,如果出现问题,首先求助的就是在线帮助。一个糟糕的在线帮助会大大地打击用户对系统的信心。因此,一个好的系统,必须要有完备的帮助体系,包括用户操作手册,实时在线帮助等。在线帮助的测试(Online Help Testing)主要用于验证系统的实时在线帮助的可用性和正确性。

2. 分析技术

在实际操作过程中,在线帮助测试可以和文档测试(或资料测试)一起进行。在进行在线帮助测试时,测试人员需要关注下面这些问题:

- 帮助文件的索引是否正确?
- 帮助文件的内容是否正确?
- 在系统运行过程中帮助能否被正常地激活?
- 在系统不同的位置激活的帮助内容与当前操作内容是否相关联?
- 帮助是否足够详细并能解决需要被解决的问题?

3. 用例设计

规范导出法。

8.2.16 数据转换测试

1. 基本概念

在实际应用环境中,经常遇到环境需要升级的问题,同时以前的数据又不能丢失,必须新的系统中能够继续使用这些数据。新系统在使用这些老数据时是否会出现问题呢?尤其当新系统采用了不同于老系统的数据格式时,这个问题尤其突出。数据转换测试(Data Conversion Testing)就是针对这个目的而进行的,该测试的目标在于验证已存在数据的转换并载入一个新的数据库是否是有效的。

2. 分析技术

在设计数据转换测试时,需要考虑的一些关键因素包括:

- 审计能力

需要有一个计划来进行转换数据前后的比较和分析以保证转换的成功。保证审计能力的技术包括文件报告、比较程序和回归测试。回归测试检查验证转换过的数据不改变业务需求或者引起系统表现的不同。

- 数据库验证

在转换之前,新的数据库需要被评审以验证它被适当的设计、能够满足业务需求,并

且支持人员和数据管理人员已经被培训过。

- 数据整理

在数据转换到新系统之前，老的数据需要被检查以验证数据中的不正确和矛盾已经被消除。

- 恢复计划

需要准备好回退步骤把系统恢复到以前的状态并且撤销转换操作。

- 同步

必须保证转换过程不会和正常的操作混杂在一起。敏感数据，例如客户数据，可能需要在转换期间被动态的修改。达到这个目的的一个方法是在系统空闲时进行转换操作。

3. 用例设计

规范导出法

8.3 系统测试过程

系统测试评价整个应用的功能和性能，是由许多测试组成的：功能测试、性能测试、可用性测试、压力测试、文档测试、安全性测试、容量测试、恢复性测试等等（参考 8.2 节内容）。类似于单元测试和集成测试，系统测试也需要遵循一定的过程。本节我们将从 Deming 的 PDCA 模型过程来讨论系统测试过程^{[8][10][11]}。在开始之前我们首先来介绍 Deming 循环。尽管该方法是由 Shewhart 提出来的，由于 Deming 把它介绍到了日本，后来就以 Deming 命名了。除了其他的 Deming 管理原则，它是日本制造工业发生改变的基础。“管理”这个词描述了不同的功能，包括政策管理、人力资源管理 and 安全管理。同样还包括组件管理、物料管理、设备管理和日期表管理等。在本节中，Deming 模型将被用到软件质量上。图 8-5 描述了 PDCA 这个过程。

在循环的计划(Plan)部分，人们定义目标，并且确定为得到这些目标而需要的条件和方法。在这个阶段，为了获取目标而需要对目标及策略进行清晰的描述。如果可能的话，一个特殊的目标应当使用数字进行文档化。同时还应当描述为获取目标而采取的方法和手段的条件及过程。

在循环的做(Do)部分，条件被创建了，为执行计划而必须的培训被执行了。每个人完全理解目标和技术是至关重要的。工人们需要被教会为完成计划而需要的过程和技能，并且完全地理解工作。这样工作才能按这些步骤进行。

在循环的检查(Check)部分，必须通过检查来确定工作是否按计划进行着，并且是否获得期望的结果。设定过程的执行必须根据条件的改变或可能出现的异常进行检查。工作的结果应当尽可能地和目标进行比较。如果检查检测到一个异常——其实际值与目标值不一致——那么异常原因的调查必须进行以防止类似的事情再次发生。有时必须对工作进行再培训，并修订过程。确定这些修改被反映并更完整地开发到下一个计划中是非常重要的。

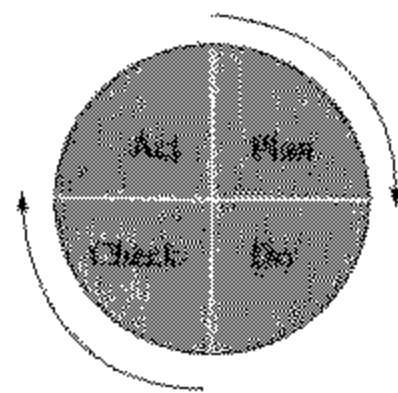


图 8-5 PDCA 循环

在循环的行动(Act)部分,如果检查揭示了工作并没有按照计划执行,或者结果并不是预计的,那么为了合适的行动必须进行适当的测量。

应用这个模型,我们可以大致安排一个系统测试的过程,该过程包括了如何准备系统测试、设计和脚本化系统测试、执行系统测试和报告测试中发现的异常。图8-6描述了这个过程。

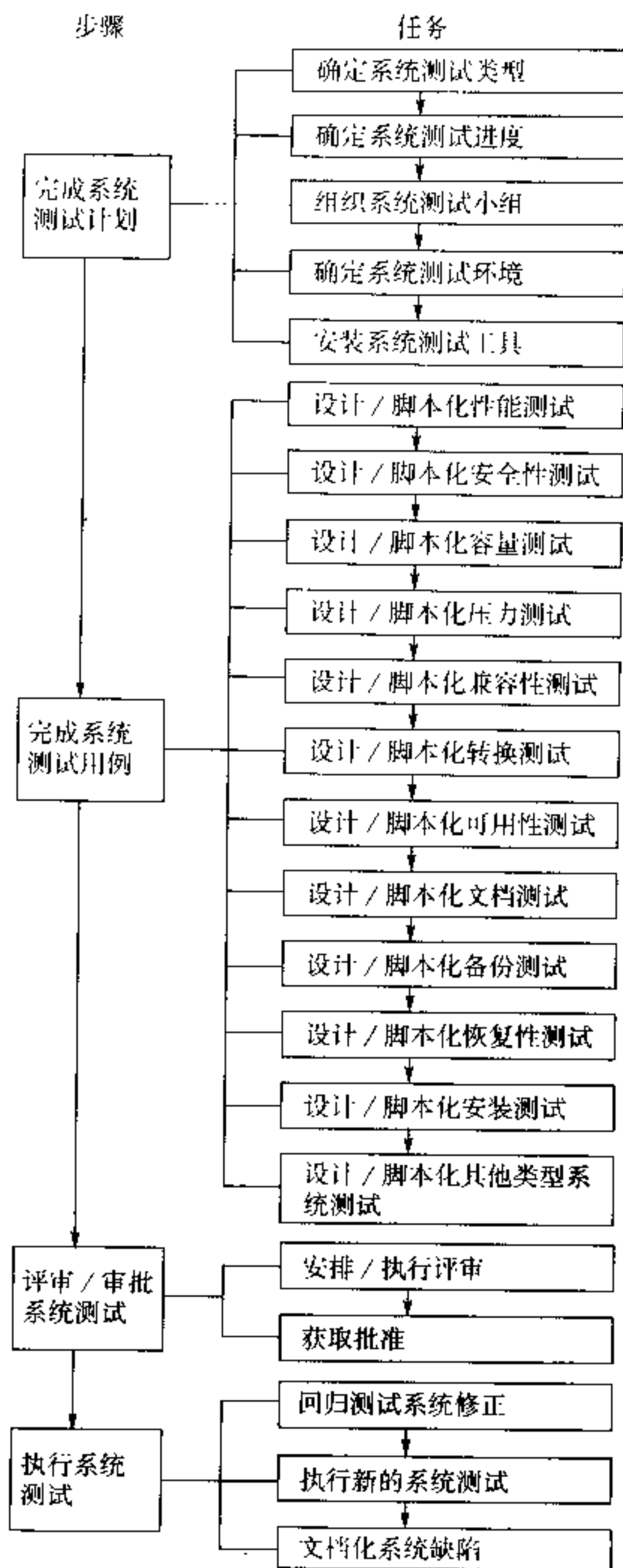


图8-6 系统测试过程

8.3.1 完成系统测试计划

从 V 模型角度来看, 系统测试计划最早可以在需求分析完成之后开始。在该计划中主要需要完成下面这些任务。

1. 决定系统测试类型

在 Deming 的 PDCA 模型中, 系统的开发遵循一个螺旋模型, 在每个螺旋期间一组系统测试部件已经被选择和执行了。当前这个任务的目的是决定系统测试类型, 这将在系统测试期间被执行。

系统测试是由一个或多个基于系统最初目标的测试组成的, 在这个任务中, 我们的目的是选择要执行的系统测试, 而不是去实现测试。最初的列表包含下面这些系统测试类型:

- 功能测试;
- 性能测试;
- 安全性测试;
- 容量测试;
- 压力测试;
- 兼容性测试;
- 转换测试;
- 可用性测试;
- 文档测试;
- 备份测试;
- 恢复性测试;
- 安装测试。

系统测试类型执行的顺序也应当在这个任务中被定义。例如, 功能测试需要最早进行, 性能测试、压力测试和容量测试可以放在一起并且在基本功能测试完成后开始进行安全性测试、备份测试和恢复性测试也可以在逻辑上分为一组进行。

最后, 可以通过测试工具进行自动化的系统测试需要被确定下来。自动化的测试提供了 3 个方面的好处: 可重复性、均衡性和增强的功能性。可重复性使得自动化测试可以被一致地执行多次。平衡性来自于测试的可重复性, 这些测试来自于先前捕获的测试和通过工具被程序化的测试, 如果没有自动化, 这是不可能的。随着应用的演化, 越来越多的功能被加入。有了自动化, 功能覆盖率可以使用测试库进行维护。

2. 确定系统测试进度

在这个任务中, 系统测试进度应当被定稿并且包括测试步骤(和可能的任务)、目标开始和目标结束日期以及相应的责任。它还应当描述系统测试如何被评审、跟踪和审批。一个系统测试进度的例子显示在表 8-3 中(同时参考甘特图模板)。

表 8-3 确定系统测试进度

测试步骤	开始日期	结束日期	职责
一般设置			
组织系统测试小组	12/1/98	12/7/98	Smith, 测试经理
确定系统测试环境	12/1/98	12/7/98	Smith, 测试经理
确定系统测试工具	12/1/98	12/10/98	Jones, 测试人员
性能测试			
设计/脚本化测试	12/11/98	12/15/98	Jones, 测试人员
测试评审	12/16/98	12/16/98	Smith, 测试经理
执行测试	12/17/98	12/22/98	Jones, 测试人员
重新测试系统缺陷	12/23/98	12/25/8	Jones, 测试人员
压力测试			
设计/脚本化测试	12/26/98	12/30/98	Jones, 测试人员
测试评审	12/31/98	12/31/98	Smith, 测试经理
执行测试	1/1/98	1/6/98	Jones, 测试人员
重新测试系统缺陷	1/7/98	1/9/98	Jones, 测试人员
容量测试			
设计/脚本化测试	1/10/98	1/14/98	Jones, 测试人员
测试评审	1/15/98	1/15/98	Smith, 测试经理
执行测试	1/16/98	1/21/98	Jones, 测试人员
重新测试系统缺陷	1/22/98	1/24/98	Jones, 测试人员
安全性测试			
设计/脚本化测试	1/25/98	1/29/98	Jones, 测试人员
测试评审	1/30/98	1/31/98	Smith, 测试经理
执行测试	2/1/98	2/6/98	Jones, 测试人员
重新测试系统缺陷	2/7/98	2/9/98	Jones, 测试人员
备份测试			
设计/脚本化测试	2/10/98	2/14/98	Jones, 测试人员
测试评审	2/15/98	2/15/98	Smith, 测试经理
执行测试	2/16/98	2/21/98	Jones, 测试人员
重新测试系统缺陷	2/22/98	2/24/98	Jones, 测试人员
恢复性测试			
设计/脚本化测试	2/25/98	2/29/98	Jones, 测试人员
测试评审	2/30/98	2/31/98	Smith, 测试经理
执行测试	3/1/98	3/6/98	Jones, 测试人员
重新测试系统缺陷	3/7/98	3/9/98	Jones, 测试人员
兼容性测试			
设计/脚本化测试	3/10/98	3/14/98	Jones, 测试人员
测试评审	3/15/98	3/15/98	Smith, 测试经理
执行测试	3/16/98	3/21/98	Jones, 测试人员

续表

测试步骤	开始日期	结束日期	职责
重新测试系统缺陷	3/22/98	3/24/98	Jones, 测试人员
转换测试			
设计/脚本化测试	4/10/98	4/14/98	Jones, 测试人员
测试评审	4/15/98	4/15/98	Smith, 测试经理
执行测试	4/16/98	4/21/98	Jones, 测试人员
重新测试系统缺陷	4/22/98	4/24/98	Jones, 测试人员
可用性测试			
设计/脚本化测试	5/10/98	5/14/98	Jones, 测试人员
测试评审	5/15/98	5/15/98	Smith, 测试经理
执行测试	5/16/98	5/21/98	Jones, 测试人员
重新测试系统缺陷	5/22/98	5/24/98	Jones, 测试人员
文档测试			
设计/脚本化测试	6/10/98	6/14/98	Jones, 测试人员
测试评审	6/15/98	6/15/98	Smith, 测试经理
执行测试	6/16/98	6/21/98	Jones, 测试人员
重新测试系统缺陷	6/22/98	6/24/98	Jones, 测试人员
安装测试			
设计/脚本化测试	7/10/98	7/14/98	Jones, 测试人员
测试评审	7/15/98	7/15/98	Smith, 测试经理
执行测试	7/16/98	7/21/98	Jones, 测试人员
重新测试系统缺陷	7/22/98	7/24/98	Jones, 测试人员

3. 组织系统测试小组

对于所有测试类型,都需要组织系统测试小组。系统测试小组的职责是设计和执行测试、评价测试结果、向开发人员报告任何缺陷和使用缺陷跟踪系统。当开发人员修正缺陷时,测试组重新测试缺陷以保证修改正确。

系统测试小组由测试经理领导,他的职责是:

- 组织测试小组;
- 建立测试环境;
- 组织测试方针、过程 and 标准;
- 保证测试准备就绪;
- 完成测试计划并且控制测试项目;
- 跟踪测试成本;
- 保证测试文档的正确和及时;
- 管理测试组成员。

4. 建立系统测试环境

在这个任务期间，系统测试环境被确定。测试环境的目的是为测试活动提供一个物理框架。测试环境需要在实现之前被建立和评审。

测试环境的主要组成部分包括物理测试设备、技术和工具。测试设备组件包括物理设置。技术组件包括硬件平台，物理网络和它们的所有部件，操作系统软件和其他软件。工具组件包括任务特殊的测试软件，例如自动化测试工具、测试库和支持软件。

需要建立测试设备和工作场所。这些可以是一个单独的工作场所配置，也可以是一个正式的实验室。无论哪种情况，测试人员尽可能靠近开发人员是很重要的。这有助于他们进行交流并完成共同的目标。系统测试工具需要被安装。

需要设置硬件和软件技术。这包括测试硬件和软件的安装并且与供应商、用户和信息技术人员进行沟通。测试硬件和与硬件供应商协调可能是需要的。通信网络需要被安装和测试。

5. 安装系统测试工具

在这个任务期间，系统测试工具被安装并验证是否准备就绪。工具测试用例和脚本的试运行应当被执行以验证测试工具已经准备好进行实际的系统测试。其他一些与工具准备相关的事情包括：

- 测试小组工具培训；
- 工具与操作系统的兼容性；
- 充足的硬盘空间；
- 最大化工具的潜能；
- 供应商工具帮助热线；
- 修改测试过程以适应工具；
- 安装最新的工具变更；
- 保证供应商合同规定。

8.3.2 完成系统测试用例

在这个步骤期间，系统测试用例被设计和脚本化。概念性的系统测试用例被转换成可重用的包含测试数据的测试脚本。在本例中我们选择的系统测试用例并不是完备的，不同的产品需要根据自己的实际情况，从业务特点、进度、成本和质量等多个角度，根据本章8.2节中的测试内容以及第9、第10章中的有关测试内容，选择自己合适的测试类型进行设计和脚本化。

8.3.3 评审/审批系统测试计划

测试计划必须经过评审和审批，因此该步骤中主要包含两个任务。

1. 安排/进行评审

系统测试计划应当在实际评审之前被很好地进行安排,并且参与者应当有测试计划的最新复制。

和任何其他评审一样,它应当包含4个部分。第一个部分是定义要讨论什么,或者“讨论我们将要讨论什么”。第二个部分是讨论细节,或者“讨论它”。第三个部分是总结,或者“讨论我们已经讨论了什么”。最后一个部分是时间点。评审者应当在这个评审期间都在场,并且如果在完成议程中所有项之前时间已经用完,那么评审者需要设定基本的规则,一个紧接着的评审会被安排。

这个任务的目的是为开发和项目发起人同意和接受系统测试计划。如果在评审期间有任何建议的修改,这些修改应当被合并到测试计划中。

2. 获得批准

在一个测试工作中,审批是关键,因为它有助于在测试、开发和发起人之间达成必要的一致意见。最好的方法是使用系统测试计划的一个正式签名过程。如果这样,可以使用管理者审批签名表格。然而,如果没有一个正式的批准过程,那么给每个参与者发送一个备忘录,名单至少包括项目经理、开发经理和发起人。在文档中附上最新的测试计划并指出他们所有反馈的意见都已经合并到计划了,并且如果你没有收到他们的信,表示他们默认了计划。最后,在一个螺旋开发环境中,系统、测试计划会随着每次迭代而演化,但是在任何一次修改中,你都要包含它们。

8.3.4 执行系统测试

1. 回归测试

如果当前这个测试不是第一个版本的测试(或者第一个螺旋内的测试),那么就需要执行这个任务。该任务的目的是在当前这个版本的系统测试中,重新测试先前系统测试周期中发现的缺陷。用到的技术是回归测试。回归测试(Regression Testing)是一个检测由于软件修改或者修正引起的不合逻辑的错误。

在软件的整个生命周期中,一套测试用例必须被维护并可用。测试用例应当足够完整以便所有的软件功能和性能能够被完整地测试。这里的问题是如何使用先前测试螺旋发现缺陷的用例。一个优秀的机制是再测试矩阵,表8-4是一个再测试矩阵的例子。

再测试矩阵(Retest Matrix)把测试用例和功能(或者程序单元)相关联起来。在矩阵中的一个检查入口表示由于功能被加强或纠正而改变后需要被重新测试的用例。没有入口表示测试不需要重新进行。再测试矩阵可以在第一个测试螺旋之前被建立,但是需要在随后的螺旋期间被维护。当功能(或者程序单元)在开发螺旋期间被修改时,在再测试矩阵中,需要检查存在的测试用例或者创建新的测试用例以为下一个测试螺旋做准备。在随后的螺旋中,经过长时间的测试,一些功能(或程序单元)可能趋于稳定,在近期内一直没有被修改过。应当考虑在测试螺旋之间删除它们的检查入口。

表 8-4 再测试矩阵

测试功能	测试用例				
	1	2	3	4	5
业务功能					
订单处理					
创建新订单	✓	✓	✓	✓	
填写订单					
编辑订单					
删除订单	✓			✓	
客户处理					
创建新客户					
编辑客户					
删除客户		✓			
财务处理					
受到客户付款		✓	✓		✓
存储付款					
付款给供应商					
写一张支票	✓	✓	✓	✓	✓
显示注册					
库存处理					
得到新供应商产品					
维护库存					
处理过去订单	✓	✓	✓	✓	✓
审计库存					
调整产品价格					
报告					
创建订单报告					
创建账户可接受报告	✓	✓	✓	✓	✓
创建账户可付款报告					
创建库存报告					

2. 执行新的系统测试

这个任务的目的是执行新的系统测试，这些测试是在先前螺旋结束时被创建的。

3. 文档化系统缺陷

在系统测试执行期间，测试的结果必须被报告到缺陷跟踪数据库中。这些缺陷一般与已经进行的单独的测试相关。然而，不同于正式的测试用例，它们通常覆盖其他的缺陷。这个任务的目的是产生一个缺陷的完整报告。如果执行步骤已经被适当地记录，那么缺陷已经被记录在缺陷跟踪数据库中了。如果缺陷已经被记录，这个步骤的目标变成收集和合

并缺陷信息。

工具可以用于根据测试执行的策略合并和记录缺陷。如果缺陷被记录在纸上，合并工作包括收集和整理这些纸。如果缺陷通过电子方式记录，搜寻特性可以很容易定位重复的缺陷。

8.4 本章小结

系统测试是对已经集成好的软件系统进行彻底的测试，以验证软件系统的正确性和性能等满足其规约所指定的要求，检查软件的行为和输出是否正确并非一项简单的任务，它被称为测试的“先知者问题”。因此，系统测试应该按照测试计划进行，其输入、输出和其他动态运行行为应该与软件规约进行对比。软件系统测试方法很多，主要有功能测试、性能测试、压力测试、容量测试等。为了进行全面的系统验证，一般需要综合多种测试方法结合进行测试。具体测试内容的选择需要根据业务的特点、进度、成本和质量等多个维度进行考虑。在本章我们介绍了16种系统测试的方法，其实在实际系统测试过程中还有一些别的测试方法，有些方法我们将在第9章和第10章有所介绍，有些方法由于只适用于某些业务领域，因此在这里就不涉及了，还有一些测试基本可以融入到上面介绍的16种测试方法中，例如数据库测试可以在功能测试、性能测试、压力测试、容量测试等多个领域内分别进行。同时在上面介绍的多种系统测试方法中，彼此之间也并不是完全没有关联的，例如功能测试和协议测试经常会在一起进行，压力测试、容量测试以及性能测试也经常混合在一起进行。具体如何做需要根据实际来判断。

系统测试不是一个突发性的测试，必须经过一个从计划到实现的过程。本章介绍的基于PDCA模型的测试过程体系必须被理解，同时结合前面单元测试过程、集成测试过程，读者应当对测试过程方面有一个比较全面的认识。有关全面介绍测试过程方面的内容可以参考笔者即将出版的《软件测试过程》一书。

第9章 可靠性与可靠性测试

可靠性测试是测试中的一个难题，本章将从工程角度比较全面地介绍可靠性和可靠性测试方面的知识。通过本章的学习，你需要掌握：

1. 什么是软件可靠性？为什么要研究软件可靠性？
2. 软件可靠性与硬件可靠性有什么区别？
3. 常用的可靠性指标有哪些？
4. 如何进行可靠性分配？
5. 了解常用的可靠性分析方法，如 FMEA、FTA、ETA、CA 以及 SCA；
6. 什么是可靠性测试？
7. 如何进行可靠性测试？
8. 了解常用的可靠性模型，包括 5 种黑盒模型与 2 种白盒模型。

随着软件应用的越来越广泛，软件可靠性也变得越来越重要。在航天领域、军事领域、核电工业领域以及医疗领域，由于软件可靠性问题而引起的灾难事件越来越多。根据美国国家宇航局 NASA 的统计，在 20 世纪 80 年代初，软件引起的故障与硬件引起的故障，其比例约为 1.1:1.0，到了 20 世纪 80 年代末，这一比例已达到 2.5:1.0。因此获得高可靠性软件已经成为这些领域最优先的任务之一，例如系统要求每年的当机时间不能超过 3s 等。这些要求只有在系统经过高可靠性设计和严格的可靠性测试之后才能被保证。测试是一个高成本的过程，需要有效的技术来提高测试的效率。尽管可靠性测试不能根除故障，但是它给系统的可靠性提供了想法。

9.1 基本概念

9.1.1 什么是软件可靠性

软件可靠性(Software Reliability)可以定义为：在规定环境，规定时间内(自然单元或时间单元)，一个系统或其功能无故障运行的可能性^[113]。

其中：

(1) 规定的环境包括硬件环境和软件环境。

- 软件环境包括允许的操作系统、应用程序、编译系统、数据库系统等。
- 硬件环境包括 CPU、内存、I/O 等。

(2) 规定的时间一般分为执行时间、日历时间和时钟时间。其中：

- 执行时间(Executing Time)是指执行一个程序所用的实际时间和中央处理器所用的时间，或者是程序处于执行过程中的一段时间。
- 日历时间(Calendar Time)指的是编年时间，包括计算机可能未运行的时间。
- 时钟时间(Clock Time)是指从程序执行开始到程序执行完毕所经历的钟表时间。

(3) 自然单元是除时间外,跟软件产品处理数目相关的单元。如运行、电话呼叫、API调用等都可以看作是一个自然单元。

从用户角度来看,软件可靠性可以分4个层面。

第1个层面:软件不出现故障;

第2个层面:软件即使出现故障,也仅对性能有部分影响,而设备的功能不受损失;

第3个层面:软件出现故障并造成部分功能受损失,但是能够很快恢复业务;

第4个层面:软件出现较大故障,并造成大部分功能受损、业务中断或瘫痪,但能够尽快恢复业务(无论是手工恢复还是自动恢复)。

对应于不同的可靠性层面,要求系统有相应的层次设计要求和维护要求,例如

对于第1个层面:要求系统能够按照充分的规范来进行设计,加强各种异常处理能力和环境适应能力等;

对于第2个层面:要求系统有较高的容错能力,使用冗余技术和备份技术等;

对于第3个层面:要求系统有很好的可测试性,能迅速隔离问题和定位问题等;

对于第4个层面:要求系统有较高的可维护性等。

9.1.2 错误、缺陷、故障和失效

在研究软件可靠性时,需要澄清几个容易混淆的概念,即:错误(Error)、缺陷(Defect)、故障(Fault)和失效(Failure)。

- 错误是指导致软件包含故障的人的行为【ISO 982.1】。
- 缺陷是指产品的异常情况。
- 故障是指引起一个功能部件不能完成所要求的功能的一种意外情况【ISO 729】。
- 失效是指功能部件执行其规定功能的能力丧失【ISO 729】。

因此一个软件的失效过程可以描述如图9-1所示。

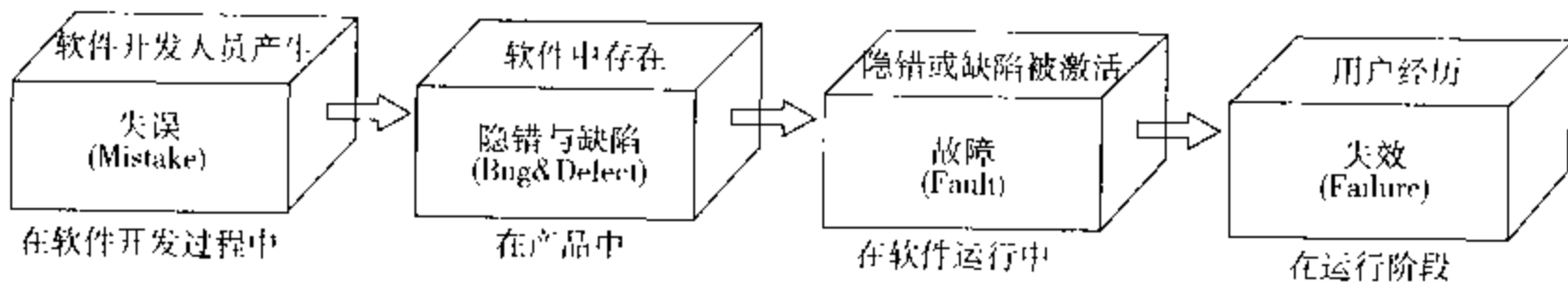


图 9-1 软件失效过程

在软件的失效过程中,其状态转移图可以描述为图9-2所示。

从失效本身的类型来分,又可以分为以下几种。

- 独立失效(Independent Failure):指不是由于另一个产品失效而引起的失效。
- 从属失效(Dependent Failure):由于另一产品失效而引起的失效。
- 系统性失效(Systematic Failure):由某一固有因素引起,以特定形式出现的失效。它只能通过修改设计、制造工艺、操作程序或其他关联因素来消除。注:无改进措施的修复性维修通常不能消除其失效原因。系统性失效可以通过模拟失效原因来诱发。
- 偶然失效(Random Failure):产品由于偶然因素引起的失效。只能通过概率或统

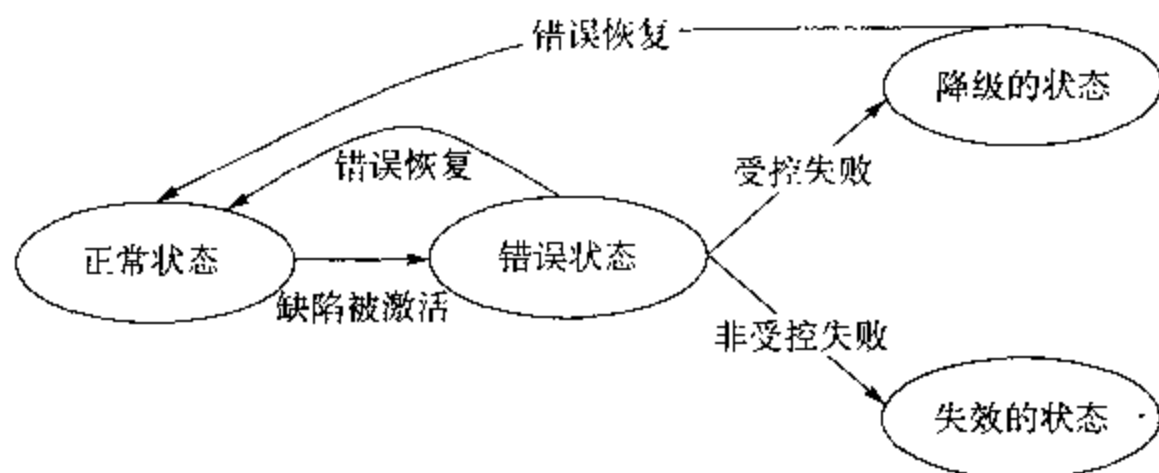


图 9-2 软件失效状态转移

计方法来预测。

- 单点失效(Single-point Failure): 引起产品失效的, 且没有冗余或替代的工作程序作为补救的局部失效。
- 间歇故障(Intermittent Failure): 产品发生故障后, 不经修理而在有限时间内自行恢复功能的故障。
- 渐变失效(Gradual Failure): 通过事前的检测或监测可以预测到的失效, 它是由于产品的规定性能随寿命单位数增加而逐渐变化引起的。对电子产品也称漂移失效(Drift Failure)。
- 致命性失效(Critical Failure): 使产品不能完成规定任务的或可能导致人或物重大损失的失效或失效组合。
- 灾难性失效(Catastrophic Failure): 导致人员伤亡或系统毁坏的失效。
- 非关联失效(Non-relevant Failure): 已经证实是未按规定的条件使用而引起的失效。或已经证实仅属某项将不采用的设计所引起的失效。
- 非责任失效(Non-chargeable Failure): 非关联失效或事先已经规定不属某组织机构责任范围内的关联失效。

9.1.3 软件可靠性指标

软件可靠性涉及到很多指标, 在此就一些关键的指标做一个解释。

1. R (Reliability) 可靠度

定义 在规定时间内无失效发生的概率。设规定的时间为 t_0 , 软件发生失效的时间是 ξ , 那么:

$$R(t_0) = P(\xi > t_0)$$

2. $MTTF$ (Mean Time To Failure) 平均失效时间

定义 $MTTF$ 是指当前时间到下一次失效时间的平均值。假设当前时间到下一次失效的时间是 ξ , ξ 具有累计概率密度函数 $F(t) = P(\xi \leq t)$, 即可靠性函数 $R(t) = 1 - F(t)$, 那么:

$$MTTF = \int_0^{\infty} R(t) dt$$

3. MTBF (Mean Time Between Failure) 平均失效间隔时间

定义 MTBF 是指两次相邻失效时间间隔的平均值。假设两次相邻失效时间间隔是 ξ , ξ 具有累计概率密度函数 $F(t) = P(\xi \leq t)$, 即可靠性函数 $R(t) = 1 - F(t)$, 那么:

$$MTBF = \int_0^{\infty} R(t) dt$$

4. MTTR (Mean Time To Repair) 平均修复时间

定义 从一次故障产生到故障恢复的间隔的平均值。

5. A (Availability) 可用度

定义 在要求的外部资源得到保证的前提下, 产品在规定的条件下和规定的时间区段内可执行规定的功能的能力。

$$A = (MTBF) / (MTBF + MTTR)$$

该可用度定义排除了失效产生后可能花费的行政时间和后勤时间, 即失效一旦产生后, 维修人员就已经在现场了。

6. DownTime: 年中中断时间

定义 产品在一年中被中断运行的时间总和。

$$DownTime = (1 - A) \times 8760 \times 60 \quad (\text{分钟} / \text{每年})$$

7. λ : 失效率

定义 产品失效的概率。

$$\lambda = 1 / MTBF$$

失效率的单位是 FIT, $1 \text{ FIT} = 10^{-9} (1/\text{小时})$

一般来说, 在进行可靠性测试时, 使用 MTTF 就可以了。但是对于投入稳定使用的、具有失效后自动恢复能力的软件系统来说, 可以选用 MTBF 参数。美国国家标准 ANSI/AIAA R-013-1992 要求最合理的故障率大约在 10^{-4} /小时左右。对于安全用计算机, 我们推荐的计算机 MTBF 大约是 1000 ~ 10000 小时。对于要求 MTBF 超过 10000 小时, 需要借助冗余技术。

图 9-3 是一个简单的计算例子。

从图 9-3 中可以得到:

产品的总运行时间 = $15 + 6 + 20 + 30 + 40 + 55 + 20 = 186$ 天 = 4464 小时;

产品发生故障的次数 = 6 次;

总的故障恢复时间 = $2 + 3 + 2.5 + 2.5 + 2.5 + 2.5 = 15$ 小时;

$MTBF = \text{产品总运行时间} / \text{总故障次数} = 4464 / 6 = 744$ 小时;

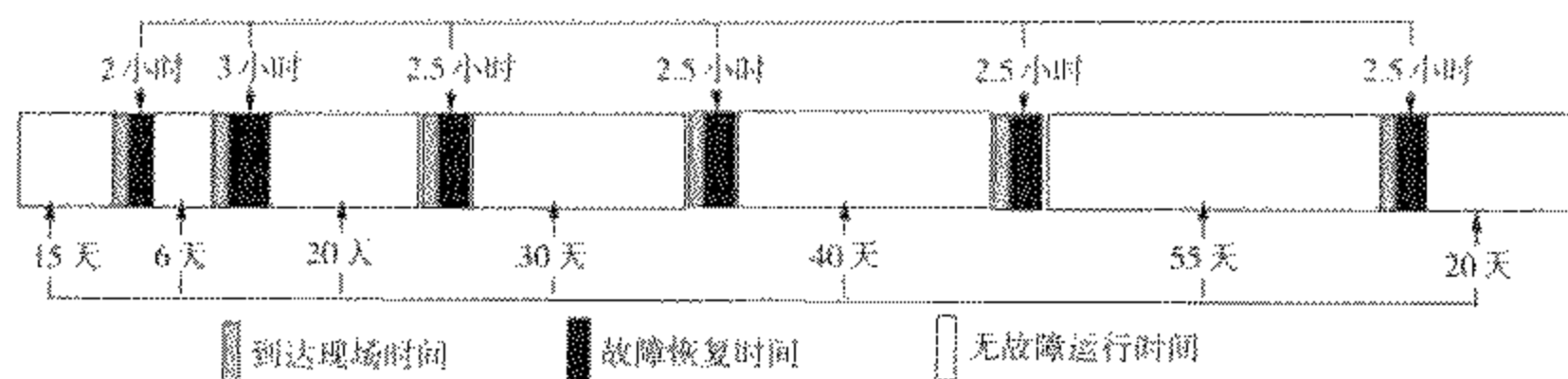


图 9-3 可靠性指标计算例子

$MTTR = \text{产品总故障时间} / \text{总故障次数} = 15 / 6 = 2.5 \text{ 小时}$;

产品的可用度 $= MTBF / (MTBF + MTTR) = 744 / (744 + 2.5) \approx 99.6651\%$;

产品的失效率 $= 1 / MTBF = 1 / 744 \approx 0.001344086 (1/\text{小时}) \approx 1344086 \text{ FIT}$;

如果需要考虑到达现场的时间(我们假设这是一个固定时间 2 小时),那么:

产品的可用度 $= MTBF / (MTBF + MTTR + MLDT) \approx 99.3988\%$

注:如果在计算时把最后一次正常运行时间去掉,或者把最后一次故障修复时间和最后一次正常运行时间去掉,在此情况下获得的软件可靠性又有所不同,且上下差别较大。这说明,同样的一套试验数据,使用不同的标准进行可靠性计算,其得到的结果也是不同的。而且这些数据你都可以认为是正确的,因为可靠性是通过已有数据推算出来的一个估计值,你很难得到一个真实的数字,这也是可靠性工程引人入胜的地方。

9.1.4 软件和硬件可靠性区别

硬件的可靠性研究已经相当成熟,软件的可靠性是从硬件的可靠性发展而来的,但是由于软件本身的特性导致软件可靠性与硬件可靠性存在较大的不同。业界关于软件可靠性的理论还在进一步完善当中。

表 9-1 显示了软硬件在修复特性上的区别,图 9-4、图 9-5 显示了软硬件在失效率曲线上的区别^[17]。

表 9-1 软硬件在修复特性上的比较

硬件	软件
存在失效不可修复的产品,例如:轮胎,电刷等产品	不存在不可修复的失效
失效后若进行完全的修复,则失效特性不变,即 $MTBF$ 不变	失效后完全修复,失效特性发生变化,即 $MTBF$ 会改变
如果失效后不修复而继续使用,将导致系统功能降级,系统的失效特性发生变化	失效后可以不修复而继续使用,并且与失效前相比,并不存在功能上的降级。如果对软件使用的操作概图也保持不变,软件的失效特性也保持不变。这是因为只要不修复,软件中导致失效的缺陷一直都存在,只要满足一定的条件,失效就会产生

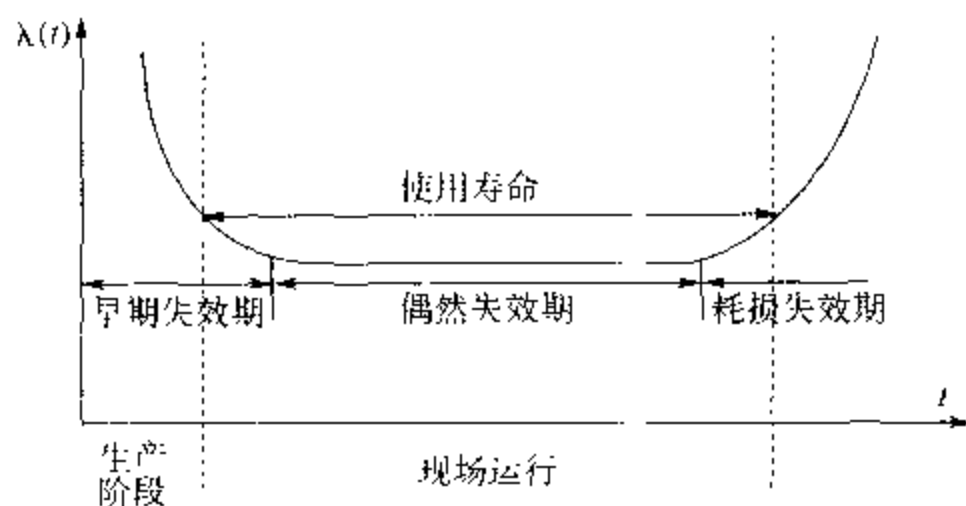


图 9-4 硬件失效曲线

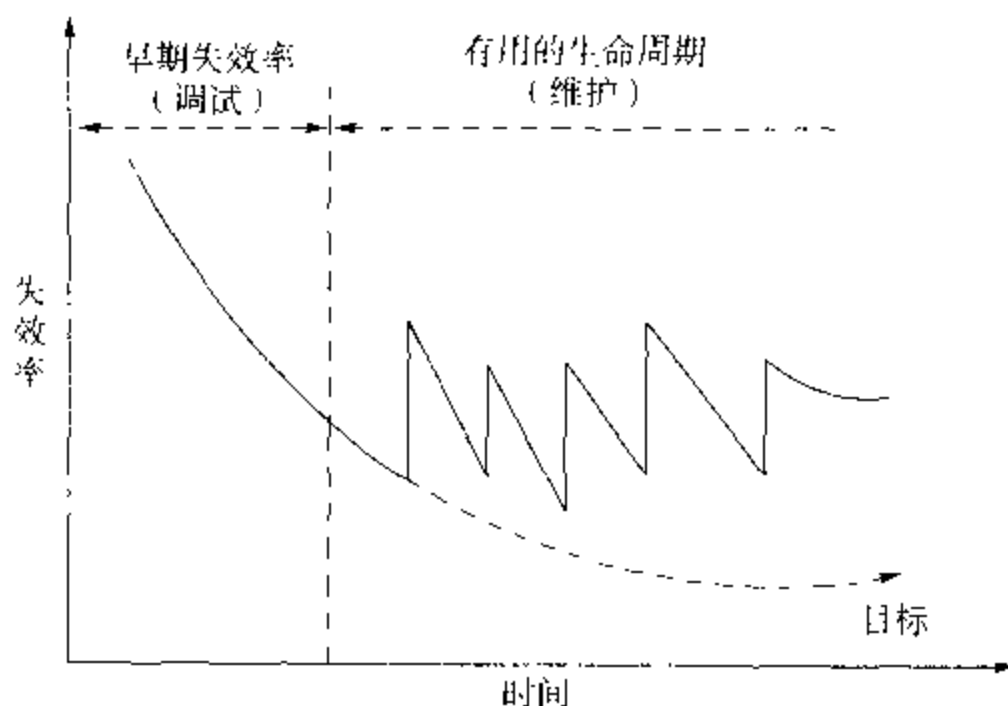


图 9-5 软件失效率曲线

9.2 可靠性指标分配

可靠性指标分配是指把系统的可靠性指标分配给系统、子系统、模块、元器件(或函数)。其主要目的是使各级设计人员明确其可靠性设计要求，并研究实现这些要求的可能性及方法。它也是可靠性试验和评估的依据。

对于电子设备，可靠性可以从整机一直分配到各个元器件。可靠性分配的目的就是使各级设计人员明确其可靠性设计要求，根据要求估计所需的人力、时间和资源，并研究实现这个要求的可能性及方法。然而对于软件来说，可靠性分配却有很大的不同，因为把可靠性分解到每一行源代码是没有意义的。对于一个系统，软件可靠性可以作为一个整体来进行考虑，它和硬件可靠性一起组成了整个系统的可靠性。它们直接的关系可以用下面的公式表示：

$$1/MTBF_{\text{系}} = 1/MTBF_{\text{硬}} + 1/MTBF_{\text{软}}$$

在硬件方面，可靠性分配时需要考虑各器件的组合方式(并联还是串联)，同时还要考虑各种加权因子(如重要性因子、复杂因子、环境因子、标准化因子、维修性因子和元

器件质量因子)。一般来说,重要的单元应分配较高的可靠性,复杂度高的单元应分配较低的可靠性,处于恶劣环境下的单元应分配较低的可靠性,技术上不成熟的单元应分配较低的可靠性。

总之,对可靠性指标的分配必须做到合理协调、技术上可行、经济上合算。分配的可靠性指标,必须进行可靠性分析,如果分配给分系统的可靠性指标因当前技术水平和条件所限,而无法实现者,必须修改方案,重新分配,直到满足要求为止。

9.3 可靠性预计

可靠性预计是可靠性指标分配的一个逆过程。它是在已知部件或以往系统的可靠性基础上推算出当前系统可能的可靠性。在产品进行可靠性预计前,必须建立产品的可靠性模型,根据产品的可靠性模型进行可靠性预计。可靠性预计的方法可以根据用途和研制阶段进行选择,数据来源应根据通用的规范或用户认可的其他数据源进行。

可靠性预计要按照从下(单元级)至上(系统级)的步骤分级进行。其步骤与可靠性指标分配相反。可靠性预计方法主要包括:计数法和应力法。

9.3.1 计数法

计数法主要用于产品总体设计阶段,此时产品的设计信息较少,可用计数法作较粗略的预计;计数法假设元器件的寿命是指数分布的,并且其失效率是恒定的。该方法将系统中各种型号和类型的元器件数目乘以相应型号元器件的基本故障率,最后把各乘积累加起来得到系统的失效率。其公式可以大致表示如下:

$$\lambda_{\text{系}} = \sum_{i=1}^n N_i (\lambda_c \cdot \pi_q)_i$$

其中:

$\lambda_{\text{系}}$ 表示总的系统失效率;

N_i 表示第 i 种单元的数量;

λ_c 表示第 i 种单元的通用失效率(一般可通过参考相关的手册或通用标准获得);

π_q 表示第 i 种单元的通用质量系数;

n 表示单元的总数。

9.3.2 应力法

应力法用于产品研制后期的详细设计阶段。在这个阶段已知所使用的元器件规格、数量、工作应力和环境、质量系数等,在使用相同元器件时,对它们的失效率、调整系数(π 系数)所做的假设应当是相同的、正确的。另外元器件应力分析法假设元器件寿命服从指数分布(即具有恒定失效率)。

使用该方法时需要考虑产品的可靠性模型,如果产品可靠性模型是串联的,或者为取

得近似值可以假设它们是串联的, 则可以把元器件失效率相加, 直接得到产品的失效率。如果产品可靠性模型中有非串联部分, 则可以先计算模型的非串联部分的等效串联失效率, 再与其他成分的元器件失效率相加。其公式大致可以表示如下:

$$\lambda_{\text{系}} = \sum_{i=1}^n N_i \left(\lambda_{ci} \cdot \prod_{j=1}^m \pi_{ij} \right)$$

其中:

$\lambda_{\text{系}}$ 表示总的系统失效率;

N_i 表示第 i 种单元的数量;

λ_{ci} 表示第 i 种单元的通用失效率;

π_{ij} 表示第 i 种单元的第 j 个 π 系数, 其中一般都会包含质量系数和环境系数;

m 表示第 i 种单元的 π 系数个数;

n 表示单元的总数。

9.4 可靠性分析方法

可靠性分析方法其目的是为了通过对系统进行可靠性分析发现系统可能出现的可靠性故障, 从而采取预防和改进措施, 以提高系统的可靠性。常见的可靠性分析方法有: 失效模式影响分析方法(FMEA: Failure Mode Effect Analysis)、严酷性分析(CA: Criticality Analysis)、故障树分析(FTA: Fault Tree Analysis)、事件树分析(ETA: Event Tree Analysis)、潜在电路分析(SCA: Sneak Circuit Analysis)等方法。

9.4.1 FMEA

故障模式影响分析(FMEA)就是在产品设计过程中, 通过对产品各组成单元潜在的各种故障模式及其对产品功能的影响进行分析, 并把每一个潜在故障模式按它的严酷程度予以分类, 提出可以采取的预防改进措施, 以提高产品可靠性的一种设计分析方法^[178]。尽管预计每一个失效模式是不现实的, 但是开发组应当尽可能广泛地制定潜在故障模式列表。

通过 FMEA 可以达到如下目的:

- 能帮助设计者和决策者从各种方案中选择满足可靠性要求的最佳方案;
- 保证所有元器件的各种故障模式及影响都经过周密考虑;
- 能找出对系统故障有重大影响的元器件和故障模式, 并分析其影响程度;
- 有助于在设计审议中对有关措施(如冗余措施)、检测设备等作出客观的评价;
- 能为进一步定量分析提供基础;
- 能为进一步更改产品设计提供资料;
- 在设计阶段评价来自客户或其他参与者的需求, 避免引入潜在的故障;
- 在设计阶段跟踪和管理潜在的风险;
- 保证任何可能产生的失效不会伤害产品或过程的顾客;

- 提高客户满意度；
- 为测试和开发提供关注点。

存在多种 FEMA 方法，其中有些方法用得比别的方法更普遍。无论在什么时候，只要失效可能对终端用户产生潜在伤害，就应当进行 FEMA 分析。常见的 FEMA 方法有^[178]：

- 针对系统级的 FEMA：关注于整个系统的功能；
- 针对设计级的 FEMA：关注于组件或子系统；
- 针对过程级的 FEMA：关注于制造和装配过程；
- 针对服务级的 FEMA：关注于服务性功能；
- 针对软件级的 FEMA：关注于软件功能。

FEMA 分析过程可以分为 4 个步骤，首先，需要根据产品的功能框图(参考图 9-6)获得产品的可靠性框图(参考图 9-7)，描述系统各功能单元的工作情况、相互影响及相互依赖关系，以便可以逐层分析故障模式产生的影响。

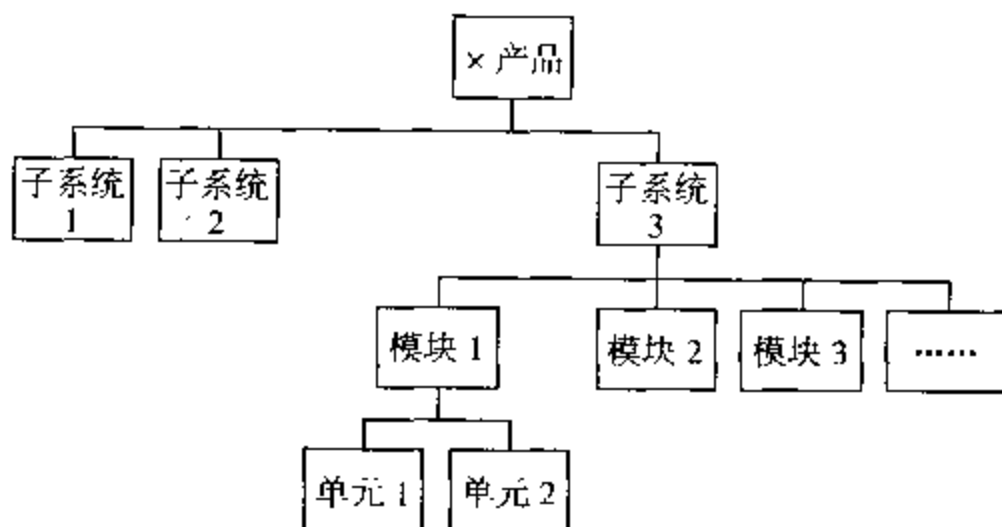


图 9-6 功能性框图

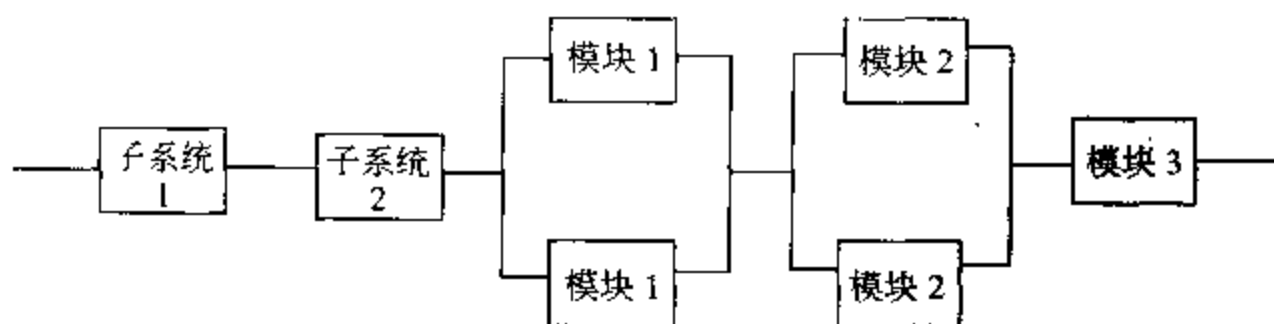


图 9-7 可靠性框图

其次，需要完成以下产品定义工作：

(1) 对产品所使用的所有不同类型单元进行编号，这主要是为了跟踪故障模式，保证故障模式分析的完整性；

(2) 对环境进行分析，明确系统使用的环境，以便确定对可靠性有影响的环境系数；

(3) 进行严酷度定义。一般我们把严酷度分为 4 级，分别是：

- 这种故障会导致整个系统崩溃或主要功能受到严重影响；
- 这种故障会导致系统主要功能受到影响、任务延误的系统轻度损坏或存在较大的故障隐患；
- 系统次要功能丧失或下降，须立即修理，但不影响系统主要功能的实现的故障；

- 部分功能下降，只需一般维护的，不对功能实现造成影响。

第二、进行FMEA分析，包括：

(1) 确定故障模式和故障原因。例如，对于光信号有无光、光功率衰减过大、时序错误(包括时间和逻辑方面的错误等)等故障；对于电信号有常高、常低、开路、短路、时序错误等故障；表9-2给出了一个对于单元级故障分析的表格样例

表 9-2 电容器故障模式分析

分类	类型	失效形式	百分比
电容器	纸/塑料薄膜电容器	短路	74
		开路	13
		参数漂移	13
	玻璃釉电容器	短路	53
		开路	25
		参数漂移	22
	云母电容器	短路	83
		开路	10
		参数漂移	7

(2) 对故障模式进行分析，明确故障检测方法(如何发现故障)，故障补偿方法(故障产生后如何处理)，定义故障的严酷度级别等，表9-3给出了一个分析的模板。

表 9-3 故障模式分析表格模板

单元编码	单元名称	功能	故障模式	故障原因	对模块的影响	对系统的最终影响	故障检测方法(已有)	检测灵敏度(已有)	补偿措施(已有)	严酷度类别	故障检测方法(建议增加)	检测灵敏度(建议增加)	补偿措施(建议增加)	严酷度类别(改进后)	需软件增加检测措施	需硬件增加检测模块
------	------	----	------	------	--------	----------	------------	-----------	----------	-------	--------------	-------------	------------	------------	-----------	-----------

最后，对分析结果进行统计。

9.4.2 CA

严酷度分析(CA)是根据每一种故障模式的严酷度类别及故障模式发生概率所产生的影响对其分类，以便全面地评价各种可能的故障模式的影响。用RPN(Risk Priority Number, 风险优先数)来定量表示，其公式如下：

$$RPN = S \times P \times D$$

其中：S——严重程度，对于I~IV类故障分别选取值100, 5, 1, 0.2；

P——故障发生的概率；

D——客户发现故障的概率，很容易发现为5，稍加注意可发现为1，不会被发现为0。一般不考虑该参数。

CA是FMEA的补充和扩展，两者经常结合起来使用，形成一种新的分析方法FMECA(Failure Mode Effect Criticality Analysis)，该方法类似于FMEA，不过却在FMEA分析表格上增加了两列：

- 严酷度总的评价

- 降低严酷度可能的活动

9.4.3 FTA

故障树分析(FTA)在产品的设计过程中,通过对可能造成产品故障的各种因素(包括硬件、软件、环境、人为因素等)进行分析,画出逻辑框图(即故障树),从而确定产品故障原因的各种可能组合方式的一种可靠性分析技术。它是用于分析大型复杂系统可靠性、安全性以及故障诊断的一个有力工具。

该方法存在两个问题:

- 不能表示实时问题;
- 不能表示系统状态或操作模式。

在使用 FTA 时,需要注意以下问题:

- 输入的可能性很小;
- 被动的组件;
- 对故障树进行定量分析(尽管可以对 FTA 进行量化,但 FTA 不是一个量化分析方法);
- 把故障树代替任何别的分析方法;
- 小心使用布尔表达式;
- 独立的失效模式与非独立的失效模式对应起来;
- 确保顶部的事件具有高优先级。

图 9-8 给出了 FTA 分析的一些符号,图 9-9 给出了一个病人监视系统故障树分析的一部分。

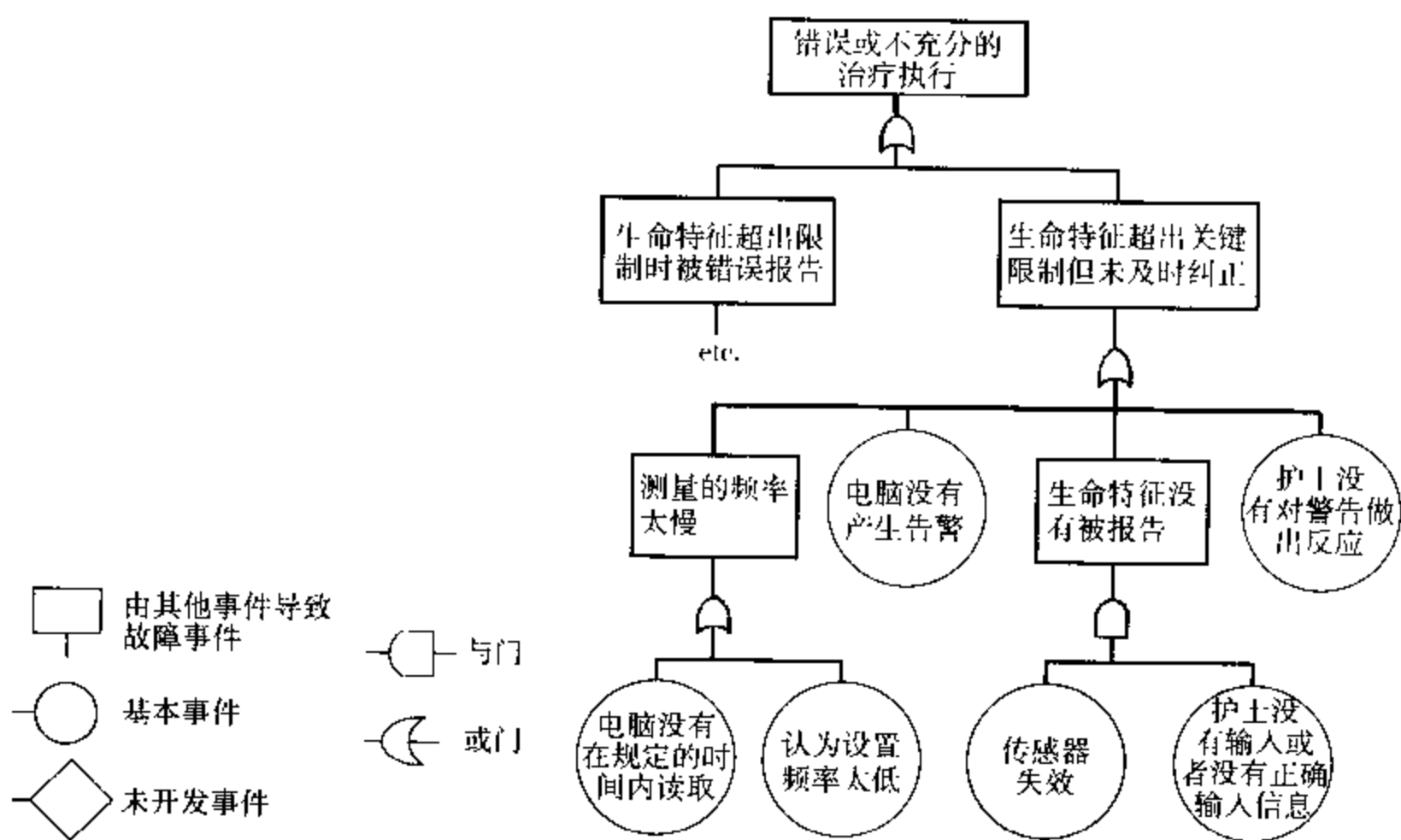


图 9-8 FTA 符号

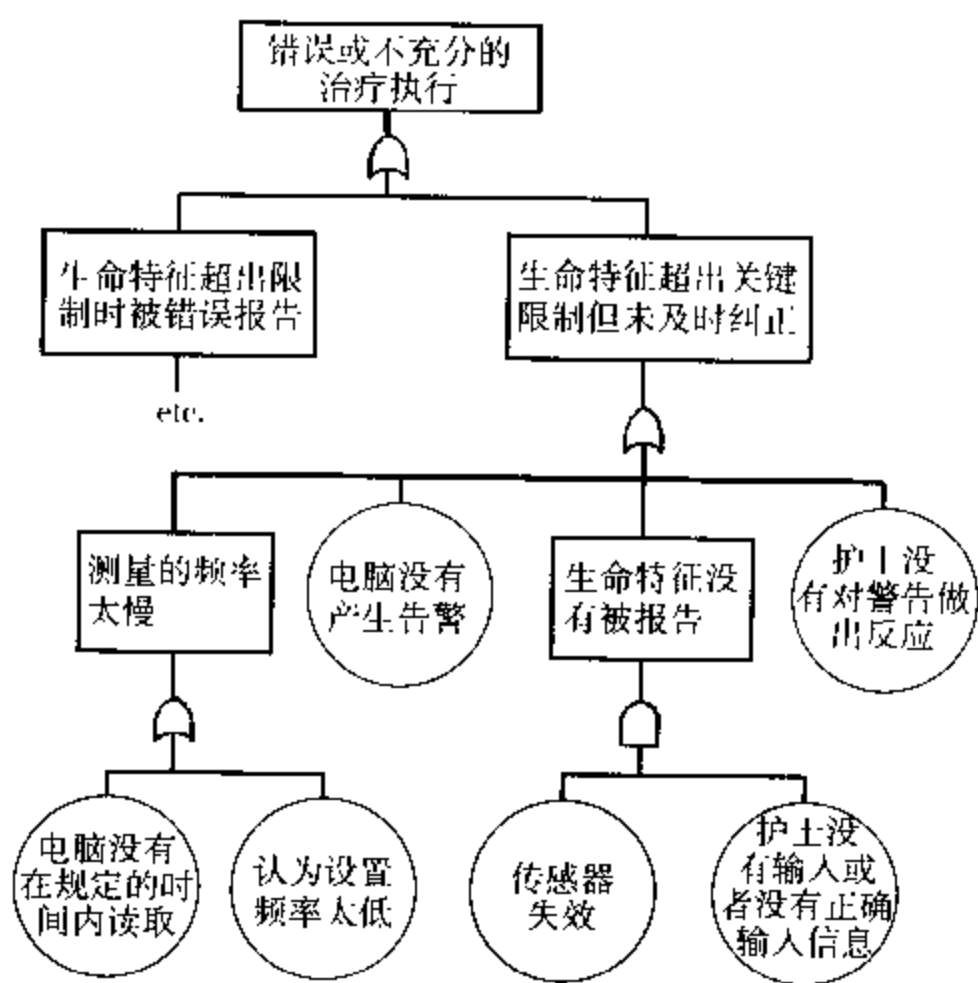


图 9-9 病人监视系统故障树分析

9.4.4 ETA

事件树分析(ETA)可以在 FTA 分析之后开始,它通过分析系统中的一个初始事件,然后考虑这个事件所有可能出现的后续事件,尤其是那些可能导致事故的事件。ETA 的起始点是那些扰乱正常系统操作的事件,接着它跟踪事件的传递,确定后继系统组件的失效,计算它们失效的可能性及可能的组合(与/或)。

ETA 的缺点是:

- 一个事件的所有可能的后继考虑起来是有困难的;
- 对于一个复杂的系统,事件树将变得非常复杂,这是因为正常和不正常的所有操作都会显示在事件树中。

图 9-10 是一个事件树的例子。

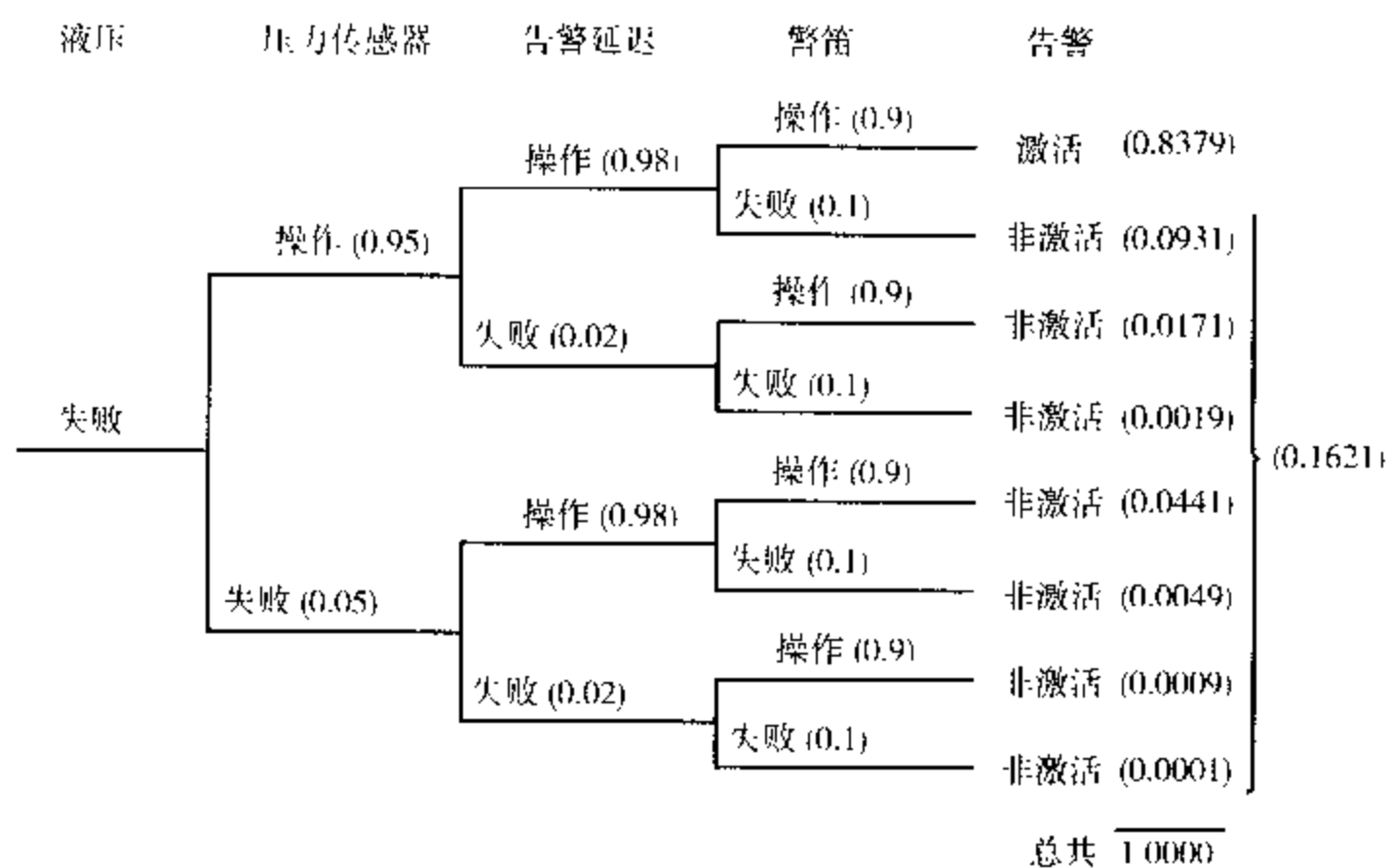


图 9-10 一个事件树的例子

9.4.5 SCA

潜在电路分析(SCA)最早是 Boeing 在 NASA 资助下开发出来的分析方法,主要用于评价电器元件^[187]。之后 Boeing 把该方法进行了改进,扩展到了计算机软件以及包含软硬件的复杂设计中,经常称为 SA(Sneak Analysis)。Boeing 把该技术应用到了 250 多个项目当中,包括商业的、NASA 的、DoD 的以及其他的一些项目。在这些项目中被确认和纠正了超过 5000 个潜在条件,为系统节省了上百万美元的损失。

SA 通过软件问题的早期确认和纠正来提高软件的可靠性,它并不提供可靠性度量。通过 Boeing 的控制试验表明,不超过 40% 的潜在条件可以通过别的分析方法发现。SA 发现的问题中有一些是非常隐蔽、潜藏很深的条件,它们只有在某些特定的操作条件下才能

被发现。这些问题使用传统的分析方法是难以检测的,甚至即使检测到也难以诊断。SA 已经被证明能够发现程序员忽略的且测试难以捕捉的问题。

为了避免传统方法的陷进,在 SA 中,详细的、代码级的软件路径被跟踪并且使用网络树的方式图形化显示出来。在软件网络树中,顺序的软件行被相应的拓扑图形替换。对于一个给定的软件代码块,每个软件网络树显示了所有的逻辑路径和指令。对于有变量被引用的每个网络树与该变量被定义的那些网络树之间进行交叉引用。判定点、标签和标签使用之间的交叉引用以及子程序和子程序调用之间的交叉引用都被显示出来,这样程序流就可以很容易地被跟踪了。这些软件树彼此之间被集成在一起并且和硬件网络树一起组成网络森林。

SA 起始于一个输出,接着向后查询有什么样的输入(静态或动态)可能会引起输出成为一个潜在的值(不在预期之内的值)。有了 SA,在设计中只有一小部分可能的输入组合需要被详细检查。那些需要进一步检查的可以通过一个精确的基于规则的过程来确认。这些被称为是潜行线索(Sneak Clue)的规则是一些历史信息的组合,这些信息收集于设计逻辑和技术,包括软件语言中的缺点或不足。这些信息需要被不断地更新,以保持永远是最新的。

SCA/SA 在可靠性分析中有着重要使用,这方面详细的资料可以参考附录 E 的书目^{[184][185][186][187][189][190][191]}。图 9-11 是一个基本拓扑图的例子。

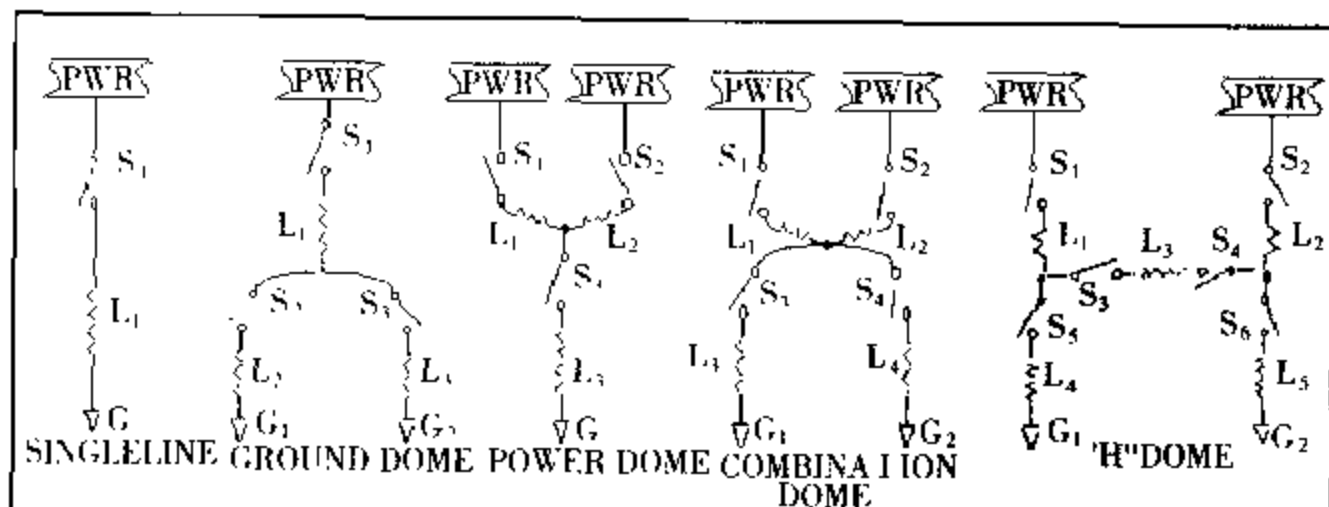


图 9-11 基本拓扑图例子

9.5 软件可靠性测试

软件可靠性测试(Software Reliability Testing)是为了达到或验证用户对软件的可靠性要求而对软件进行的测试;通过测试发现并纠正软件中的缺陷,提高其可靠性水平,并验证它是否达到了用户的可靠性要求。软件可靠性测试能有效地暴露在实际使用过程中影响可靠性要求的软件缺陷,最先暴露的是发生概率较高的缺陷,然后是发生概率较低的缺陷。软件可靠性测试的概念可以通过图 9-12 来表示。

就像本文第 1 章所说明的一样,测试并不能保证软件的可靠性,软件可靠性是设计出来的而不是测试出来的。软件可靠性测试的主要目的是验证软件可靠性指标是否被满足。它通过在测试过程中收集测试数据并使用可靠性模型来计算软件的可靠性指标。

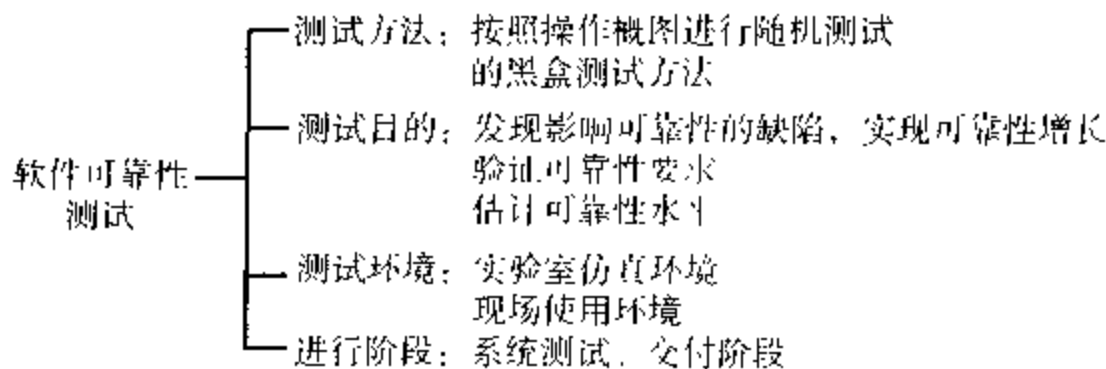


图 9-12 软件可靠性测试概念

软件可靠性测试是从硬件可靠性测试发展而来的。硬件可靠性测试的理论已经发展相当成熟，然而关于软件可靠性方面的测试理论还处在发展阶段。许多事实表明，在计算机系统中，软件可靠性水平一般要比硬件可靠性水平低。因此，为了保证系统的可靠性水平，不仅要对硬件可靠性提出要求，还必须相应地对软件可靠性提出要求。

9.5.1 可靠性测试流程

软件可靠性测试流程可以通过图 9-13 所示。

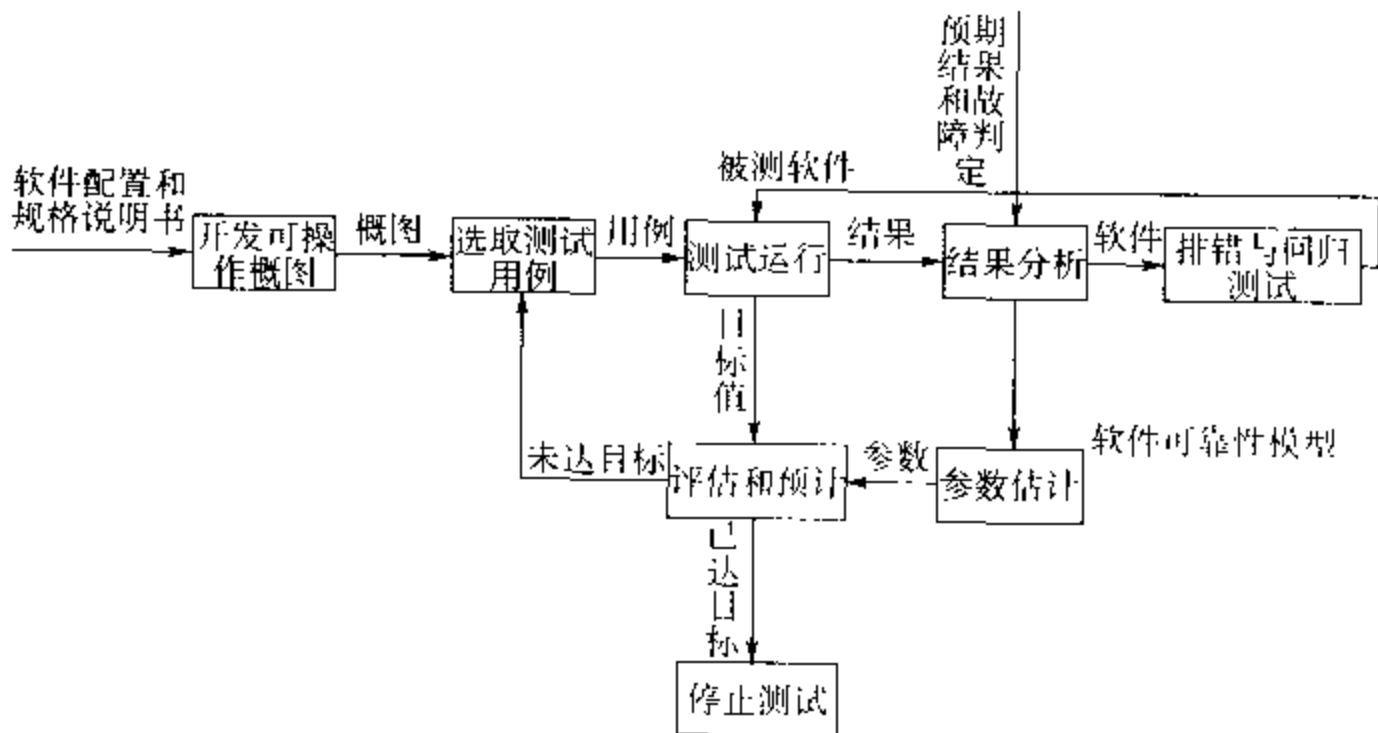


图 9-13 软件可靠性测试流程图

在软件可靠性测试中有 3 个主要的环节，它们分别是：

- 根据用户实际使用软件的方式，开发软件可操作概图，生成测试用例；
- 开发软件可靠性测试的环境，使得被测软件系统能够在该环境下被“控制”并测试；
- 对测试的结果数据进行量化分析，并得出系统的可靠性验证结果或预计系统的可靠性指标。

1. 可操作概图

软件可靠性测试的一个主要特点是按照用户实际使用软件的方式来测试软件，软件的

可操作概图(Operational Profile)是定量描述用户实际如何使用软件的一个方法。在该方法中除了要充分了解用户如何使用软件的各种模式和各种功能,完成这些功能相应的输入变量,还需要了解用户在使用软件时这些模式和功能出现的概率^[172]。这些信息大部分来自于软件开发文档、需求规格说明书和接口文档等资料。因此这要求测试人员能够充分与用户沟通,收集系统实际被使用的历史数据信息并进行充分的分析。系统模式和功能划分越完整,概率越准确,那么构造出来的这个视图就越接近实际使用情况。

一个软件的可操作概图按照一个层次的结构被组织,自顶向下把用户使用软件的输入空间划分为系统模式概图,再把系统模式概图划分为功能概图,最后划分到运行概图。图9-14表示了这一划分关系。

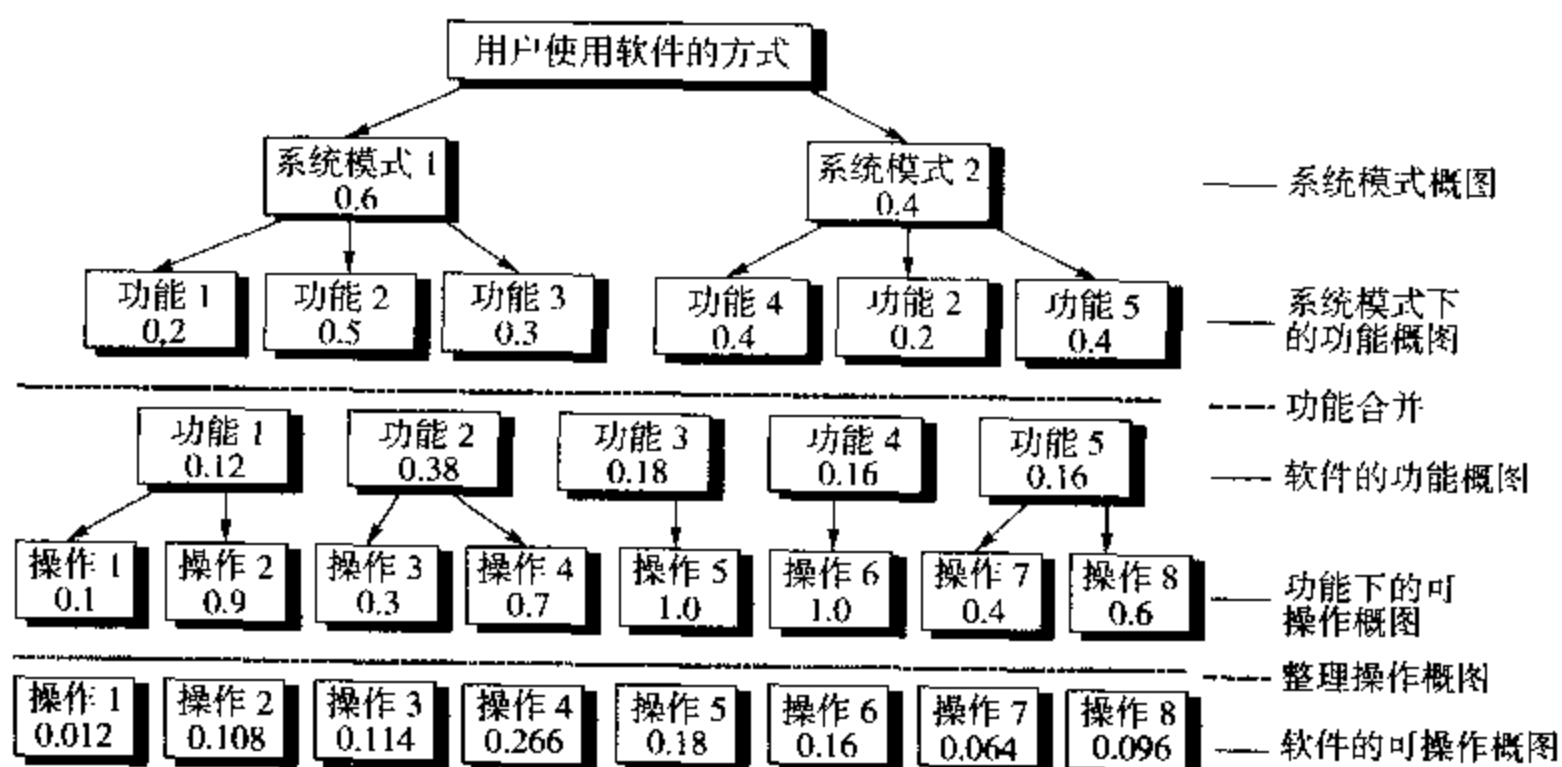


图 9-14 系统模式概图、功能概图和可操作概图关系

2. 测试用例生成

测试用例是根据可操作概图生成的。在可操作概图中规定了每个输入变量的取值范围,并且认为变量的取值在这个范围中是均匀分布或者分段均匀分布。这是因为软件可靠性测试是一种随机测试,测试用例的选取方式是随机的。

可靠性测试用例的准确性是基于对输入空间的正确分析及使用概率的合理判断。对于输入空间的分析包括了对输入变量取值范围的分析以及输入变量相互之间的约束关系分析。

根据用户需求,若一个软件功能由一个或多个软件基本功能组成,可单独完成一个完整的任务,并可进行单独的测试,那么称这个软件功能为一个约功能。所有约功能输入变量输入时刻的定义域和取值的定义域的直积构成了约功能的输入子空间,所有约功能的输入子空间构成了软件的输入空间^[192]。对于实时软件,输入变量的输入时刻可以是周期性变化的或非周期定时的,也可以是随机时刻输入的。输入变量的取值可以是确定的,也可以是随机的。输入变量的特性及其与之对应的取值区间和概率分布变化的规律分析可以参考表9-4。对于输入变量之间的约束关系是错综复杂的,在此就不详细描述了,具体可以参考附录E^{[165][192]}中的资料。

表 9-4 输入变量特性分析

取值特性	输入时间要求	取值区间	概率分布
确定性	周期性	输入周期、各输入周期的具体取值或周期的函数关系	
	非周期定时	各给定输入时刻的具体取值	
	随机时刻	输入时刻的取值区间、在随机给定的输入时刻的具体取值	输入时刻的概率分布
非确定性	周期性	输入周期、各输入周期的取值区间	各输入周期内各取值区间上对应的概率分布
	非周期定时	各给定输入时刻的取值区间	在各给定输入时刻取值区间上对应的概率分布
	随机时刻	变量输入时刻的取值区间、变量在各输入时刻的取值区间	变量输入时刻的概率分布、变量在各输入时刻取值区间对应的概率分布

3. 测试环境

为了完成软件可靠性测试，一个完备的仿真测试环境应具备以下基本功能：

- 初始化。包括自检、系统配置、确立测试方案等；
- 测试准备。包括准备测试时需要的数据，预运行以保证定时器的启动和 I/O 服务的启动等；
- 测试执行。包括测试数据仿真、产生激励信号、测试结果接受、测试过程监控以及测试结果处理；
- 测试结果分析处理。包括回放显示测试数据和测试结果数据，包括确定显示内容、显示方式、回显时的缩放比例等；对测试结果数据进行比较和分析；
- 测试文档管理。负责对测试的各类文档进行管理，生成测试报告等工作；
- 系统维护管理。包括记录平台测试的配置和测试方案，生成带有时间标签的配置文件，记录工作日志，记录系统故障，修复系统故障，进行系统软件完好性检查等。

4. 量化分析

识别需要收集的数据并且建立确保收集数据完整准确的机制，是软件可靠性工程实施中最重要的一环之一。开发组织一般都具备收集软件可靠性工程应用中所需要数据的能力，所研究的每一项软件开发工作都有记录，并且跟踪在测试阶段及运行中观测到的失效数据的机制。大部分项目还要求测试人员在测试阶段做工作记录。如果运用得当，这些数据收集机制有益于当前通用的许多可靠性模型所使用的形式，并提供精确的失效数据。然而，这项工作将使产品开发花费一定的成本，因此国内很多公司基本上都没有把它做起来。

可靠性建模是可靠性估计过程的基础，它确定产品是否达到了可靠性指标，是否可以发行。它要求根据系统测试中收集的失效数据（如失效报告数据和测试时间），运用可靠性模型估计作为测试时间函数的各种产品可靠性参量。有几种相互依赖的估计参量对产品可靠性所做的描述是等价的，它们一般包括产品失效率（单位时间的失效数目）随测试时

间的变化、到测试时刻 t 的平均失效时间 ($MTTF$)。这些可靠性估计参量可以为产品质量管理提供以下有用信息:

- 系统测试结束时产品的可靠性;
- 达到产品可靠性指标尚需的(额外)测试时间;
- 通过测试获得的可靠性的增长(如测试开始时的失效率与测试结束时的比);
- 系统测试值之外的可靠性预测值。例如,如果系统测试已经完成,此值可以是产品的现场可靠性;如果系统测试尚未完成,此值可以是对系统测试终止时可靠性的预计值。

现有的可靠性模型有 40 多种,它们的选择和应用问题并不复杂。应有选择合适模型的原则和统计学方法。经验表明,考虑 40 多种模型中的十几种就足够了,特别是在它们已被软件工具所实现的情况下。在下面一节我们将介绍一些比较典型的可靠性模型。

应用这些统计学模型,在测试阶段可以得到关于可靠性的“最佳”估计。而后,这些估计值被用于校正现场试运行中的可靠性数值,根据这些数值确定可靠性指标是否得到满足。这个过程可能要重复进行,因为一旦可靠性指标未得到满足,就需要再次测试。在操作概图开发得不够详尽的情况下,应用测试压缩因子(Test compression factor)有助于现场可靠性的估计。测试压缩因子定义为在现场试运行阶段与在测试阶段覆盖整个程序输入空间的运行时间之比。因为测试时测试人员总是极力搜索整个输入空间,寻找苛刻的运行条件来“考验”软件,而现场试运行中只是按正常状态执行软件,所以测试压缩因子代表了运行比测试时观测到的失效率减少的程度(可靠性增强程度)。

最后,经过校正的现场可靠性预测值在现场运行中与实际观测值作对比,以验证其有效性。这种有效性验证不仅可以确定可靠性估计的准确程度及应给予的信任程度,而且可以为软件可靠性工程实施过程提供反馈信息,促进实施过程的改进和参数的调整,例如,可以确认模型的有效性,确定可靠性增强幅度、优化测试压缩因子等。

9.5.2 可靠性模型介绍

软件可靠性模型用于在开发早期预计软件的可靠性,它主要关注于缺陷的去除情况。软件可靠性模型包括 3 个部分:假设、可靠性因子以及把可靠性与这些因子相关联的数学函数。通过软件可靠性模型,可以在项目的测试阶段评价开发的状态,监视软件的操作性能,控制设计的变更和新特性的引入。

在本节,我们将从黑盒和白盒两个方面来介绍一些比较流行的可靠性模型^[174]。在黑盒方面,我们重点介绍 5 个模型:Jelinski-Moranda 的故障分离模型(De-eutrophication Model),Goel-Okumoto 的 NHPP 模型(Non-Homogeneous Poisson Process Model),Musa 的基本执行时间模型(Basic Execution Time Model),增强的 NHPP 模型(the Enhanced NHPP Model)以及 Littlewood-Verrall 的贝叶斯判定模型(Bayesian Model)。对于相同的数据,不同的模型可以得到不同的结果,有些结果可能大相径庭。这往往是因为不同的模型基于的假设条件不同而造成的,关于这些模型的一些比较资料在网上可以找到,也可以参考附录 E 的^[195]。

在白盒方面,我们主要介绍 Krishna-murthy 和 Mathur 的基于路径的模型(Path-Based

Model) 和 Gokhale et al. 的基于状态的模型(State-Based Model)。关于这些模型的一个详细比较可以参考附录 E 的^[196]。

1. 一些数学术语

请见表 9-5 所示。

表 9-5 一些常见术语

术语	解释
$M(t)$	到时间 t 发现的缺陷总数
$\mu(t)$	软件可靠性增长模型(SRGM: Software Reliability Growth Model)的平均值函数, 它表示该模型估计的到时间 t 期望的缺陷的数量。因此我们有 $\mu(t_i) = E[M(t_i)]$
$\lambda(t)$	缺陷强度, 由平均值函数派生出来的。因此我们有 $\lambda(t) = \mu'(t)$
$Z(\Delta t/t_{i-1})$	软件的机会几率, 它表示第 i 个错误在给定的 $t_{i-1} + \Delta t$ 时间内出现的可能密度, 其中第 $(i-1)$ 个错误出现在 t_{i-1}
$z(t)$	每个故障的机会几率, 它表示了一个还没有被激活的故障在其被激活的时候会立刻引起一个失效的概率。这个术语在许多模型中经常被假设为一个常量(φ)
N	软件在提交测试之前出现的最初缺陷数量

2. 故障分离模型

Jelinski-Moranda 的故障分离模型(J-M 模型)是最早开发出来的可靠性模型之一^[191]。这个模型假设:

- 在代码提交测试之前, 代码中的原始故障是个固定值 N , 但不知 N 是多少;
- 失效之间没有关系, 并且失效之间的时间是独立的且呈指数分布的随机变量;
- 失效产生后, 故障的去除是及时的且不会引入任何新的故障到被测系统中;
- 每个故障的机会几率 $z(t)$ 在时间上是不变的一个常量(φ)。此外, 每个故障在引起失效方面是等效的。

这个假设导致软件在第 $i-1$ 个缺陷被去除之后的机会几率是和软件中遗留缺陷数量成比例关系。因此可以得到如下公式:

$$Z(\Delta t/t_{i-1}) = \varphi(N - M(t_{i-1}))$$

这个模型的平均值函数和故障强度函数就变成:

$$\mu(t) = N(1 - e^{-\varphi t})$$

$$\lambda(t) = N\varphi e^{-\varphi t} = \varphi(N - \mu(t))$$

从这个模型中得到的软件可靠性可以表示成如下公式:

$$R(t_i) = e^{-\varphi(N-(i-1)t_i}$$

这个模型需要两个故障之间的时间值。

3. NHPP 模型

Goel-Okumoto 的 NHPP 模型(G-O 模型)与 J-M 模型的假设略有不同^[175]。两个假设之间的最大不同是在时间 t 观察到的期望的故障数量遵循泊松(Poisson)分布, 有一个带边界

且非递减的均值函数 $\mu(t)$ 。在无限时间内观察到的故障的一个期望值是一个有限的值 N 。

这个模型还做了下面这些假设：

- 出现在 $(t, t + \Delta t)$ 的软件失效数量和期望的未检测到的失效数量 $N - \mu(t)$ 成正比；
- 在内部失效间隔 $(0, t_1)(t_1, t_2) \cdots (t_{n-1}, t_n)$ 内检测到的失效数量之间没有联系；
- 每个故障的机会几率 $z(t)$ 在时间上是不变的一个常量(φ)；
- 当故障被检测到时，缺陷的排除是及时且完好的(不会引入新错误)。

假设导致了下面的均值函数用于表示在时间 t 被观察到的期望的失效数：

$$\mu(t) = N(1 - e^{-\varphi t})$$

$$\lambda(t) = N\varphi e^{-\varphi t} = \varphi(N - \mu(t))$$

该模型的可靠性可以使用与 J-M 一样的公式，因此可以用上面两个公式替换 J-M 模型中相应的公式。

4. 基本执行时间模型

Musa 的基本执行时间模型是第一个试图在测试执行时结合一个时间变化的测试工作量来反映软件可靠性增长^[194]。这个模型也是一个 NHPP 模型，并且假设在一个时间 t 被观测到的失效的数量是有限的，遵循泊松分布。这个模型的一个突出特性是 t 表示“执行时间”，该执行时间使用引起失效的测试的实际 CPU 时间来表示。

这个模型做出的一些重要假设如下：

- 两个失效之间的时间是呈幂数分布；
- 每个故障的机会几率是常量(φ)。

该模型需要失效的实际时间，因为这些时间被用在模型的参数计算中。该模型应用到失效数据上获得的失效强度是：

$$\lambda(t) = NB\varphi e^{-B\varphi t}$$

其中 N 具有 G-O 模型中相同的解释。 B 表示故障减少因子，它是一个与故障率相关的常量。上面的公式还可以表示成下面的形式：

$$\lambda(t) = \varphi(N - \mu(t)) = Kf(N - \mu(t))$$

其中 φ 被公式化成线性执行频率 f 的产品，且故障暴露率是 K ，它可以被解释为在程序的一个线性执行期间，代码中遗留的故障引起失效数量的一个平均值。该参数还被假设为与时间无关的^[176]。

这个模型的一个有趣扩展是对数泊松执行时间模型，其中，失效期望的数量是一个泊松随机变量和一个 CPU 时间 t 的对数函数，且有一个因子确定失效强度的衰弱。

5. 增强的 NHPP 模型

增强的 NHPP 模型是对于有限失效 NHPP 模型的一个统一框架^[195]，即其他有边界均值函数的 NHPP 是增强 NHPP(ENHPP)模型的一个特例。该模型在它的分析公式中明确地包含了随时间变化的测试覆盖率和不完整的故障检测。

这个模型中的测试覆盖率被定义成对一个测试敏感的潜在故障场所占总潜在故障场所数的比例。潜在故障场所指的是指“程序结构或功能元素实体，这个实体的易感染性被认为对建立软件产品运算一致性很关键”^[195]。

这个模型做出下列假设：

- 故障被一致地分布在所有潜在故障场所中；
- 当一个故障场所在时间 t 变得敏感时，其故障被检测的可能性是 $c_d(t) = K$ (一个常量)，表示故障检测覆盖率；
- 故障的排除是及时且完好的(不会引入新错误)。

这个模型的均值函数是：

$$\mu(t) = c(t)N$$

其中， $c(t)$ 是一个随时间变化的测试覆盖率函数， N 是在全覆盖率中期望被暴露的故障数量。这个模型的失效强度公式如下：

$$z(t) = c'(t)(1 - c(t))^{-1}$$

$$\lambda(t) = z(t)(N - \mu(t))$$

其中， $z(t)$ 是一个随时间变化的每个故障的机会几率。这个模型允许有缺陷的覆盖率想法被包含到可靠性估计当中。不同的覆盖率函数分布导致不同的 NHPP 模型，即，G-O 模型，Yamada S-shaped 模型等。这个模型获得的可靠性可以表示成如下公式：

$$R(t/s) = e^{-NK(c(s+t) - c(s))}$$

其中 s 是最后一个失效的时间， t 是上一个失效被检测到的时间。这个模型的主要优点是可以作为 NHPP 模型的一个统一框架。此外，每个故障机会几率的依赖性惟一依赖于随时间变化的测试覆盖率，而忽略其他影响因子，诸如这样一些事实：在检测到所有缺陷时，完全的覆盖率并不一定能够达到；在没有获得任何覆盖率增长时，也有可能检测到故障。

6. 贝叶斯判定模型

贝叶斯软件可靠性增长模型认为可靠性在已被检测的故障数量和无故障操作的上下文之内增长。此外，在没有失效数据情况下，贝叶斯模型认为模型参数有一个优先的分布，它反应了对基于历史的未知数据的判断^[196]。

Littlewood-Verrall 模型是贝叶斯判定模型的一个例子，它假定失效之间的时间是独立指数随机变量，拥有一个参数 ξ_i , $i = 1, 2, \dots, n$ ，该参数本身还包含参数 $\psi(i)$ 和 α ，分别表示程序员质量和任务难度。该模型有一个优先的伽马分布。从该模型观察到的失效强度公式如下(使用线性形式替换了 $\psi(i)$)：

$$\lambda(t) = (\alpha - 1)(N^2 + 2B\varphi(\alpha - 1))^{-1/2}$$

其中 B 表示故障缩减因子，类似于 Musa 的基本执行时间模型。这个模型需要在失效产生之间进行调整以便从先前分布中获得较后的分布。

7. 基于路径的模型

Krishnamurthy 和 Mathur 的基于路径的模型最主要的假设是组件的可靠性是已知的。

通过执行针对特定软件的可靠性测试用例, 获得用例对应的所有路径可靠性估计, 然后使用该模型求得平均值, 从而获得软件可靠性的估计。在实际操作中, 可以从软件的测试或模拟中进行执行跟踪, 获得不同路径上的组件顺序, 从而得到软件的“构架”^[198]。该构架对 K-M 模型是非常关键的。

如果每个组件有可靠性 R_i , 且假设每个组件的失效是独立的, 那么一个跟踪 $M(P, t)$ 的路径可靠性为 R_p , 其中 P 是被测程序, t 是一个可靠性测试用例, 属于测试套 T , 且 P 在执行 t 时, 经过了一组组件 m 。因此得到公式如下:

$$R_p = \prod_{m \in M(p, t)} R_i$$

整个系统的可靠性 R_{sys} 是在测试套 T 上所有路径可靠性的平均值, 公式如下:

$$R_{sys} = \sum \frac{R_p}{|T|}$$

组件间的依赖性和大循环的影响导致沿一条跟踪路径上的组件出现多次的可能性, 这使得组件不是独立的。考虑沿着一条相同路径的一个组件出现的次数是 $k(k > 0)$, 我们就可以对这类情况建模了, 其中把 k 看成是组件独立的程度。 k 值越大, 路径的估计值就越小, 系统的可靠性也越小。

Yacoub, Cukic 和 Ammar 扩展了这个模型, 使用了一个算法来估计路径可靠性, 一个穿越树的算法扩展了代表软件结构的所有图形分支。具体可以参考附录 E 的【200】。

8. 基于状态的模型

Gokhale 的基于状态的模型有个特点, 即它假设组件的可靠性要么已经获得, 要么可以通过使用 ENHPP SRGM 来确定^[175]。另一个假设是要估计可靠性的应用是一个终端应用。Gokhale 观察到如果模块间的控制转换被假设是一个马尔可夫 (Markov) 过程, 那么描述整个结构的软件控制流图可以直接映射到一个离散时间或持续时间的马尔可夫链中, 且在软件结构和马尔可夫链之间有一一对应关系。

组件的可靠性可以从 ENHPP 模型获得, 具体如下:

$$R_i = e^{-\int_0^{V_i T_i} \lambda_i(t) dt}$$

其中: V_i 是访问模块 i 期望的数量并且 $\lambda(t)$ 是依赖于失效强度的时间, 并且 $V_i T_i$ 是花费在一个模块上的总时间。整个系统的可靠性可以通过下面公式计算:

$$R_{sys} = \prod_i R_i$$

基于状态的其他模型还有: Cheung model^[198], Kubat model^[201] 以及 Laprie model^[202]。

9.5.3 一个可靠性数据分析例子

1. 原始缺陷数据

表 9-6 包含了某产品在测试期间收集到的失效数据。测试总共执行了 19.9 小时, 总共发现了 84 个故障。

表 9-6 原始缺陷数据

错误序号	上次失效后时间	严重程度	错误序号	上次失效后时间	严重程度
1	7.98E+00	1	43	1.72E+01	1
2	4.43E+00	1	44	4.07E+01	1
3	4.62E+00	1	45	7.67E+00	1
4	9.23E+00	1	46	9.42E+00	1
5	1.72E+01	1	47	1.87E+01	1
6	4.15E+00	1	48	7.28E+00	1
7	1.16E+01	1	49	1.55E+01	1
8	9.95E+00	1	50	7.44E+01	1
9	1.95E+00	1	51	1.19E+01	1
10	2.83E+00	1	52	3.02E+00	1
11	1.95E+00	1	53	2.48E+01	1
12	2.12E+01	1	54	1.26E+01	1
13	7.82E+00	1	55	5.26E+01	1
14	1.96E+01	1	56	3.53E+01	1
15	1.16E+01	1	57	1.47E+01	1
16	3.18E+01	1	58	3.40E+01	1
17	2.25E+00	1	59	2.47E+01	1
18	4.62E+00	1	60	9.32E+00	1
19	9.93E+00	1	61	8.17E+00	1
20	1.26E+01	1	62	9.88E+00	1
21	7.28E+00	1	63	2.95E+01	1
22	3.72E+01	1	64	1.42E+00	1
23	7.28E+00	1	65	4.73E+01	1
24	5.67E+00	1	66	3.55E+00	1
25	6.75E+00	1	67	3.11E+01	1
26	8.92E+00	1	68	8.17E+00	1
27	4.62E+00	1	69	2.48E+01	1
28	6.05E+00	1	70	7.20E+01	1
29	8.70E+00	1	71	2.36E+01	1
30	1.02E+01	1	72	1.71E+01	1
31	4.62E+00	1	73	9.15E+01	1
32	2.17E+01	1	74	2.53E+01	1
33	1.37E+01	1	75	5.47E+01	1
34	3.55E+00	1	76	4.53E+01	1
35	2.70E+01	1	77	3.63E+01	1
35	2.70E+01	1	77	3.63E+01	1
36	2.67E+01	1	78	5.84E+01	1
37	4.97E+00	1	79	1.21E+01	1

续表

错误序号	上次失效后时间	严重程度	错误序号	上次失效后时间	严重程度
38	1.46E+01	1	80	3.27E+01	1
39	1.03E+01	1	81	6.63E+01	1
40	4.40E+01	1	82	1.82E+01	1
41	8.33E-02	1	83	4.08E+00	1
42	2.48E+00	1	84	1.99E+01	1

2. 失效间隔时间曲线

如图 9-15 所示。

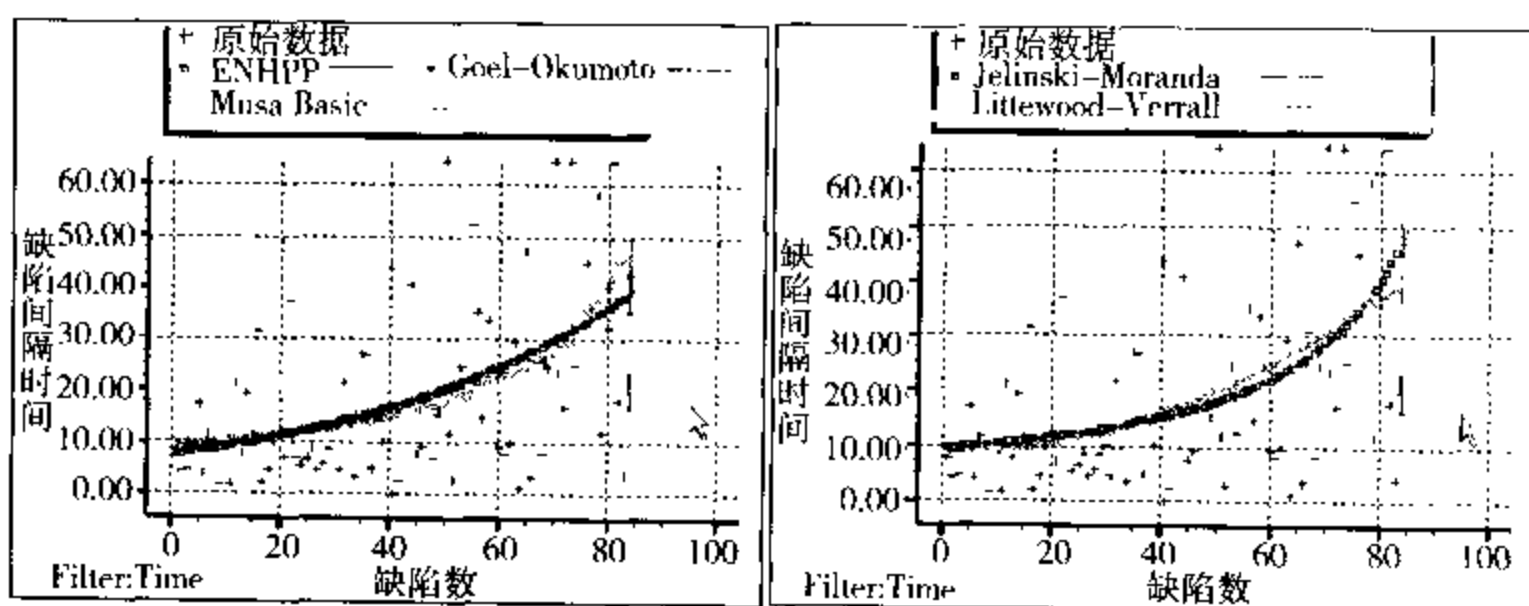


图 9-15 失效间隔时间-缺陷数曲线

3. 累计缺陷曲线

如图 9-16 所示。

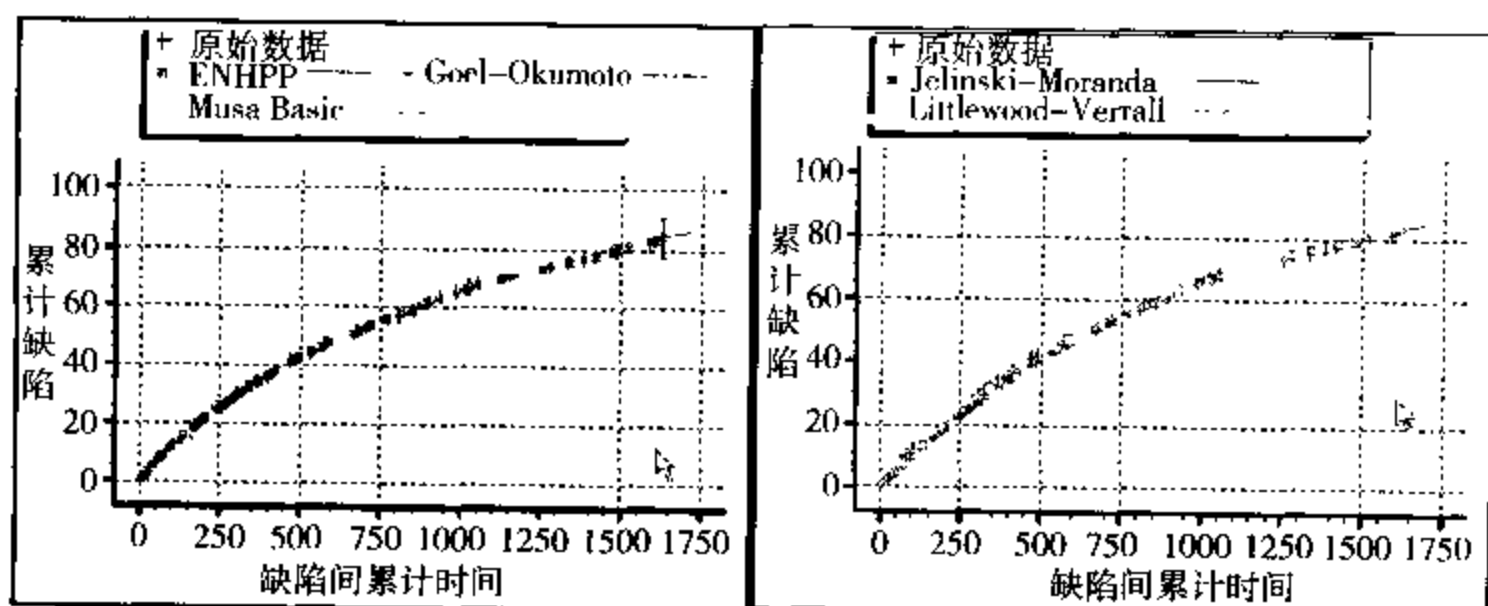


图 9-16 累计缺陷曲线

4. 缺陷密度曲线

如图 9-17 所示。

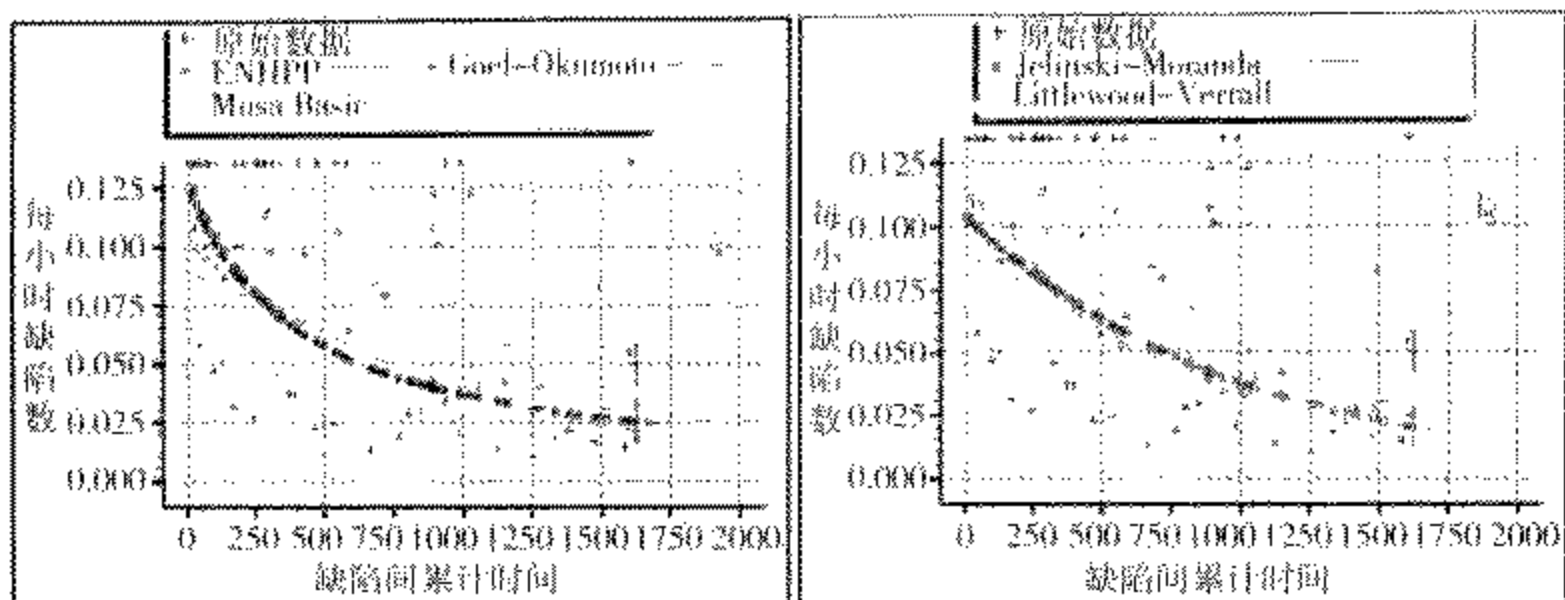


图 9-17 缺陷密度缺陷

9.6 软件可靠性工程

软件可靠性的研究发展已经形成了一套工程体系，这就是软件可靠性工程 (Software Reliability Engineering)。我们知道软件的可靠性不是测试出来的，而是设计出来的。软件可靠性工程就是研究如何在过程上保证产品的软件可靠性，它把可靠性的工作划分到了软件开发的整个生命周期。

可靠性工程内容包含了上面所提到的所有内容，从可靠性分析到可靠性设计直到最后的可靠性测试。

从整个软件开发生命周期看，软件可靠性活动大致可以用表 9-7 来表示。

9.7 可靠性标准和可靠性工具

9.7.1 可靠性标准

目前国际上比较通用的可靠性标准及手册主要包括：

- ANSI/AIAA R-013
- MIL-STD-882C
- RTCA DO-178B
- IEEE-Std-730
- MIL-HDBK 338, Electronic Reliability Design Handbook
- MIL-HDBK-217, Reliability Prediction of Electronic Equipment

- MIL-HDBK-781“Reliability Test Methods, Plans and Environments.”
- IEEE-Guide-982.1 and.2
- NHB 1700.1 Preliminary Hazards Analysis
- NASA STD 8719.13A Software Safety
- NASA GB 1740.13 Guidebook for Safety Critical Software

表 9-7 开发过程中的可靠性活动

生命周期阶段	可靠性活动	例子
需求阶段	从系统可靠性需求中产生软件可靠性目标	在一个给定的里程碑中对于一个指定的场景的最大缺陷率 在可接受测试中的最大缺陷率
	准备可靠性验证计划	估计测试用例数量
	现实性评价	和先前项目获得的可靠性进行比较
概要设计阶段	进行软件可靠性分配	在指定的场景中分析每个组件的参与程度
	选择合适的软件可靠性工具	评价所有支持软件可靠性的开发工具 获得额外的工具和培训
	为目标的获得做计划	在里程碑前根据每个组件产生获得的缺陷率
	把软件可靠性活动和系统可靠性集成到一起	通用的或兼容的工具和模型，兼容的可靠性度量
编码和详细设计阶段	填充可靠性模型	基于组件规模和用法估计可靠性
	为软件可靠性问题评审测试计划	为获得软件可靠性目标，分析测试用例的数量和类型、以及与场景的兼容性
	评审单元测试	分析遇到的错误和失效的类型，寻找共同的原因
	开始可靠性分析活动	为 FMEA, FTA, SCA 等分析方法建立框架
测试阶段	参与失效评审委员会 (Failure Review Board; FRB)	评价对可靠性重要的数据 评价 FRACAS (Failure and Corrective Action System) 格式，用于给委员会进行原因评估
	评审测试准备度	确定测试问题不会阻碍软件可靠性数据收集
	可靠性数据分析	评估软件可靠性目标满足度
	可靠性趋势分析	分析可靠性增长目标的获得
	失效原因分析	委员会进行原因评估并明确要求纠正活动
运行阶段	参与失效评审委员会	类似在测试中的活动
	失效数据分析	确定来自于先前系统和来自于测试的数据的一致性；明确要求纠正活动

9.7.2 可靠性工具

软件可靠性活动(包括测试)需要借助一定的可靠性工具,业界常用的可靠性工具可以参考表9-8的内容。

表 9-8 可靠性工具列表

工具名称	供应商	联系方式
Relex Software 7.0	Relex Software Corp.	Relex Asia/MoaSoft Corporation BackAm B/D suite 700 Karak-Dong 123 Sonpa-Ku Seoul Phone: (82)-2-420-3203 FAX: (82)-2-407-3511 Home Page: www.relexasia.co.kr E-Mail: info@relexasia.co.kr
RAM-Commander	SoHaR Incorporated Incorporated	SoHaR Incorporated 5731 W Slauson Ave. , Suite 175 Culver City, CA 90230 Telephone: 1-310-338-0990 Fax: 1-310-338-0999 E-Mail: info@sohar.com Home Page: http://www.sohar.com
FMEA Processor	SoHaR Incorporated	SoHaR Incorporated 5731 W Slauson Ave. , Suite 175 Culver City, CA 90230 Telephone: 1-310-338-0990 Fax: 1-310-338-0999 E-Mail: info@sohar.com Home Page: http://www.sohar.com
Sneak Circuit Analysis Tool	SoHaR Incorporated	SoHaR Incorporated 5731 W Slauson Ave. , Suite 175 Culver City, CA 90230 Telephone: 1-310-338-0990 Fax: 1-310-338-0999 E-Mail: info@sohar.com Home Page: http://www.sohar.com
RelQuest	Questa Computing Ltd	"Coppertrees" Forest Road Effingham LEATHERHEAD KT24 5HE United Kingdom Telephone: +44 1483 283408 Fax: +44 7957 398508 E-Mail: andrewj@andrewj.com Home Page: http://www.andrewj.com

续表

工具名称	供应商	联系方式
Lincoln Reliability Improvement Tool (LRIT)	Lincoln Technology Corporation	Lincoln Technology Corporation #116, 17704-103 Avenue Edmonton, Alberta Canada, T5S 1J9 Phone: (780)444-8820 Fax: (780)484-1005 E-mail: lincoln@lincolntechnology.com. Home Page: http://www.lincolntechnology.com
FRestimate	SoftRel	Sugar Land, Texas USA E-mail: sales@softrel.com Home Page: http://www.softrel.com
Terminal Reliability Calculation Tool	UMass College of Engineering	Home Page: http://www-unix.ccs.umass.edu/~droychow/termRel.html
Quanterion Automated Reliability Toolkit (QuART)	Rome Laboratory Quanterion Solutions Incorporated	811 Court Street Utica NY 13502-4096 Phone: 315.732.0097 Toll free: 877.808.0097 Fax: 315.732.3261 Home Page: quanterion.com E-mail: qinfo@quanterion.com
CRAX	The Center for System Reliability of Sandia National Laboratories	Robert M. Cranwell Telephone: 505-844-8368 FAX: 505-844-3321 Postal address: MS0746 PO Box 5800 Albuquerque, NM 87185-0746 FedEx deliveries: 1515 Eubank SE, Albuquerque, NM E-mail: drobin@sandia.gov Home Page: reliability.sandia.gov

9.8 本章小结

软件的可靠性已经变得越来越重要,那么什么是软件可靠性呢?软件可靠性可以定义为:在规定环境,规定时间内,一个系统或其功能无故障运行的可能性。可靠性的研究已经发展成一门工程知识,称为可靠性工程。一个产品的可靠性需要依赖整个过程的保证,从可靠性需求分析,到可靠性指标分配和设计,再到可靠性预计、可靠性分析,最后到可靠性测试和可靠性数据分析。

可靠性指标的分配是指把系统的可靠性指标按一定的层次和级别分配到子系统、模块及单元。可靠性分配的一个关键是系统构架的设计。可靠性预计是在已知单元的可靠性基础上初步估计出系统可能的可靠性,常用的方法有计数法和应力法。通过可靠性预计,可以预先知道系统的可靠性还离目标有多远。可靠性分析和可靠性预计可以结合在一起进

行,可靠性分析主要通过一定的分析手段,发现系统中可能影响可靠性的故障,从而改进系统的可靠性。常用的可靠性分析方法包括FEMA、FTA、CA、ETA、SCA等。

可靠性测试是从验证的角度出发,检验系统的可靠性是否达到预期的目标,同时给出当前系统可能的可靠性增长情况。可靠性测试需要从用户角度出发,模拟用户实际使用系统的情况,设计出系统的可操作视图,在此基础上,根据输入空间的属性及依赖关系导出测试用例,然后在仿真的环境或真实的环境下执行测试用例并记录测试的数据。对可靠性测试来说,最关键的测试数据包括失效间隔时间、失效修复时间、失效数量、失效级别等。根据获得的测试数据,应用可靠性模型,可以得到系统的失效率及可靠性增长趋势。常用的可靠性模型可以从黑盒(占主要地位)和白盒两个角度出发。黑盒方面的可靠性模型包括Musa基本执行模型,Jelinski-Moranda的故障分离模型,Goel-Okumoto的NHPP模型,增强的NHPP模型以及Littlewood-Verrall的贝叶斯判定模型。白盒方面的可靠性模型包括Krishna-murthy和Mathur的基于路径的模型和Gokhale et al.的基于状态的模型。业界流行的可靠性模型还有很多种,不同的可靠性模型其依赖的假设条件也不同,适用范围也不同,因此对于一个产品,其所适合使用的可靠性模型需要根据实际出发,尽可能选择与可靠性模型假设条件相近的模型。

第 10 章 其他专项性测试

前面两章介绍了很多系统测试的方法，以及其他一些类似的测试，有些测试可以归入到系统测试中，有些测试可以归入到系统测试后的验收测试中，有些测试可能无法明确地划分在哪个阶段进行。在本章，将介绍这些测试。通过本章的学习。

你应当了解以下的概念：

1. 可接受性测试有哪些特点？
2. 为什么要进行 Alpha 测试？
3. Alpha 测试和 Beta 测试有什么区别？
4. 如何进行配置测试？
5. 为什么要进行 2000 年测试？
6. 如何选择回归测试的范围？

10.1 可接受性测试

用户可接受性测试(User Acceptance Testing)又叫验收测试。在通过了内部系统测试及软件配置审查之后，就可以开始该项测试了。验收测试是以用户为主的测试。软件开发人员和 QA(质量保证)人员也应参加。由用户参加设计测试用例，使用用户界面输入测试数据，并分析测试的输出结果，一般使用生产中的实际数据进行测试。在测试过程中，除了考虑软件的功能和性能外，还应对软件的可移植性、兼容性、可维护性、错误的恢复功能等进行确认。验收测试实际上是对整个测试计划进行一种“走读(Walk-through)”。

用户可接受测试具有下列特点^[203]：

- 必须要有用户参与，且以用户为主；
- 可接受测试在不同的组织之间，随目标的不同及工作量的不同而不同；
- 在软件开发过程当中，可接受测试是最容易变化的一个测试；
- 用户可接受测试只有按照既定的目标进行时才能有真正的效果；
- 很少有组织能够真正理解如何处理测试中人的问题，他们同时还缺乏必要的培训来进行计划和执行一个有效的可接受性测试。

一般来说，可接受测试可以采用如下形式：

- 使用一个演示版，它是一个基本的完整系统的一个走读。这个走读可能是交互的，但用户通常在系统正式产品化之前，并不去确认缺陷；如果你只想让用户看看系统是个什么样子，那么该形式是很适合的；
- 重复先前阶段的一些测试，包括单元测试、集成测试和系统测试。更为普通的是

可接受测试作为系统测试的一个子集进行^[8]。无论使用哪种级别,在这个形式上的测试主要关注于软件在已定义需求下执行的正确性。重复开发已经执行过的测试是冗余的,并且通常会迷失可接受测试的目标;

- 从业务视角出发进行的建设性测试。用另一种话说,就是“这个软件能否在我们日常的业务环境下工作?”,该形式对于想要确定系统是否满足业务需要是最合适的。

为了成功地进行可接受性测试,可以使用如下方法:

- 保证问题和决定通知到每个人。对于确认问题并及时解决问题来说,良好的沟通是关键。当人员不能及时跟上测试问题时,他们就会根据不完整的信息做出决定;
- 通过一种双赢的模式解决冲突。任何其他解决冲突的方式最终会对人家都产生不利。用户应当感到组织内部对于他们的问题正在积极解决,同时开发人员也不应当对一些小的问题大动干戈;
- 对可接受性测试进行计划,这样有利于控制和重复;
- 关心你的用户测试人员。记住这些人员可能还有别的事情。每天把他们捆绑到测试上可能会导致他们筋疲力尽;
- 对于一些有争议的决定需要向上传递。用户测试人员不应当被要求做出一些政策性决定,这些问题应当提交给更高管理者。

10.2 Alpha 测试

Alpha 测试(α 测试)和 Beta 测试(β 测试)是产品在正式发布前经常需要进行的两个不同类型的测试。 α 测试,有时也称为室内测试(In-house Testing),是由一个用户在开发环境下进行的测试,也可以是开发机构内部的用户在模拟实际操作环境下进行的测试。软件在一个自然设置状态下使用。开发者坐在用户旁,随时记下错误情况和使用中的问题。这是在受控制的环境下进行的测试。

Alpha 测试的目的应当事先清晰地传递给每个参与者^[204]。Alpha 测试中的每个参与者在 Alpha 测试结束时提供一个反馈。在这个测试期间,项目经理应该向参与者介绍一些项目的历史背景知识。项目的设计人员应当在测试期间提供协助,并给出测试的一般规则。

Alpha 测试主要用于发现下面一些问题:

- 主要的概念性缺陷或者与主题不协调的地方;
- 发现与功能需求和项目规格不符合的地方;
- 发现在拼写、标点以及习惯用法方面的错误(针对 GUI);
- 发现图形的位置错误(针对 GUI);
- 发现不准确、不清晰或者不完整的图形(针对 GUI);
- 发现不完整或不准确的标题(针对 GUI)。

在进行一个 Alpha 测试时,需要注意以下事项。

(1) 在起始的时候, 必须明确你现在正在进行一个 Alpha 测试并且你希望做一些修改。

(2) 告诉 Alpha 测试参与者需要遵循下面一些基本原则:

- 时刻记录下对于系统的建议, 建议应该足够详细, 以便能指导修改;
- 以一定的指令顺序进行 Alpha 测试, 在时间不足的情况下, 可以提醒参与者关注系统最关键的地方;
- 尽可能地要求建议或改进而不是简单的接受批评。从项目成员那边获取协助修改的承诺。

(3) 保证有人记录下了各种评注以帮助项目组成员在修订的时候能够记起曾做出的决定。这些评注一般可以分为 3 个类型:

- 必须进行的变更。这些一般是属于错误, 并且会在正式发布的版本中被纠正;
- 有效性变更。这些主要是属于内容性变更, 一般是细化某些提示信息或帮助信息;
- 改进性变更。这些建议并不在最初的需求或项目规格中, 但是有了将更好。一般这些变更会被安排在下一个版本中。

(4) 如果你在是否进行变更的讨论中陷入了泥潭, 那么就应当把该决定推迟。但需要记得后面还需要回到该决定上。

10.3 Beta 测试

Beta 测试(β 测试)是由软件的多个用户在一个或多个用户的实际使用环境下进行的测试。这些用户是与公司签定了支持产品预发行合同的外部客户, 他们要求使用该产品, 并愿意返回有关错误信息给开发者。与 α 测试不同的是, 开发者通常不在测试现场。因而, β 测试是在开发者无法控制的环境下进行的软件现场应用。

作为 Beta 测试人员, 需要愿意使用有错误的软件, 并且这些错误可能会使计算机崩溃。同时 Beta 测试人员需要理解, 在测试过程中可能没有或者很少能从 Beta 程序中获得技术支持。

Beta 测试仅仅是测试, 而不是对软件的评价。作为软件的开发者, 愿意尽最大努力修改所有被反馈的缺陷, 但是对于那些没有反馈的缺陷一般能做的事情很少。并且对于该测试参与者来说, 参加 Beta 测试并不是无偿长期获得一个软件的方法, 因为 Beta 测试的软件一般会在测试期过后就作废了。Beta 测试的好处包括:

- 参与者可以比别人先一步看到软件的新特性;
- 参与者可以发现一些值得怀疑的错误;
- 作为检测出那些错误的结果, 软件将变得更好;
- 通过 Beta 测试人员的反馈, 可能会影响到以后开发的方式。

一般软件公司提供 Beta 测试的时候, 主要通过两种不同的途径: 公共 Beta 和私有 Beta。公共 Beta 程序允许每个人员可以访问这个软件, 有时可以自由下载, 有时只需付少量的钱就可以购买到产品的光盘。而私有 Beta 被限制在一小部分人当中, 这些人在有协议

的公司或受控的环境中。私有 Beta 测试人员要愿意积极报告缺陷并反馈信息。

现在公共 Beta 方式已经越来越广泛,最典型的例子是微软的 Beta 软件测试。随着互联网的普及,越来越多的公司把软件的测试依赖在 Beta 测试上,然而,广泛的 Beta 测试并不能完全替代实验室内的系统测试,这主要基于以下几个原因:

- Beta 测试人员不是专业的测试人员,很难发现一些深层次的问题,更多的是停留在使用性方面的问题上;
- 由于 Beta 测试是不受控的,因此无法了解 Beta 测试人员实际是如何操作系统的,有很多 Beta 测试人员反馈的问题是由于使用不当而引起的;
- 对于一些细小的问题, Beta 测试人员往往不愿意反馈;
- 有些 Beta 测试人员往往不是为了测试软件而参与,而是为了评价软件或获得软件而参与测试,并且当他们发现软件中存在一些重要缺陷时,并不是积极反馈,而是私下决定不再购买该软件;
- Beta 测试人员反馈的问题信息很简单,经常不能指导问题的修改,开发人员往往需要花费更多的精力去定位问题。

10.4 标杆测试

系统指标能够描述该产品的基本特性的性能,该指标也可以称为性能指标。系统指标在系统设计初期就会提出来,但是最终产品详细指标如何必须通过严格的测试才可以得到。要根据系统稳定性测试模型,结合系统运行的实际情况对系统进行指标测试或标杆测试(Benchmark Testing)。

系统标杆测试的基本概念可以分为两部分:

- 在系统基本配置或最优化配置条件下,通过测试工具等模拟系统环境和提供单一或标准负荷模型,从而得到系统各种表征特性的指标,进一步可以验证系统需求和设计规格中的指标是否达到;
- 在多任务并接近实际网上运行等复杂条件下,由于受 CPU、内存、存储器、通道、网络、系统配置等资源的影响而测试出系统性能在各方面潜在的低效和限制,比如系统瓶颈,系统指标上限。

对于纯软件产品的标杆测试不多,这类测试一般对制造类产品使用得比较多,例如:电信类产品。该类产品性能方面的国际国内标准随产品的不同而不同,设计指标测试时必须熟悉这些标准,遵照标准、系统需求和系统设计规格中的性能指标来进行测试。

标杆测试最好在系统稳定性测试之后进行,因为系统稳定性测试可能会发现系统的缺陷,而该问题严重影响了系统性能而不得不更改原始配置和设计模型。这些变更导致了数据配置的优化、设计的优化甚至设计的完全更新。为了得到真实系统性能指标就不能过早地开展指标测试活动。

10.5 配置测试

配置测试(Configuration Testing)验证系统在不同的系统配置下能否正确工作,这些配置包括:软件、硬件、网络等。

例如,要设计一个网站,为了测试网站的页面能够在不同的配置上可以被终端用户正确浏览,因此需要进行各类配置测试。在当今最新的网页开发环境中,有超过 7000 种可能的配置,这些配置涉及到:操作系统、浏览器、窗口大小和其他一些特性。表 10-1 是一个简单的例子。

表 10-1 一个配置例子

操作系统	浏览器	窗口大小	Cookies	Java	内存
Windows 95	NetScape 4.7X	640 × 480	Yes	Enabled	16MB
Windows 98	NetScape 5	800 × 600	No	Disabled	32MB
Windows NT	NetScape 6.2	1024 × 768			128MB
Windows ME	IE 4				256MB
Windows 2K	IE 5				384MB
Windows 2KP	IE 6				512MB
Windows XP	AOL 4				1024MB
MAC	AOL 5				
Linux	AOL 6				
	AOL 7				
	WebTV				

网页的内容可能会在任何不同的配置下被浏览。尽管一些配置能够完全按照你所计划的那样工作,而其他有些配置可能会出一些小问题,糟糕的是有些配置可能会出现和你设想的完全不同的结果。

配置测试有时经常会与兼容性测试或安装测试一起进行。

10.6 外场测试

外场测试(Site Testing)主要用于验证系统在实际使用环境中能够正常工作。这类测试一般在一些大型的系统中经常用到,比如电信系统、航天产品等。

外场测试是一个比较昂贵的测试,需要投入大量的人力和物力。而且场点的选择要具有一定的代表性。这类测试一般需要有专门的外场测试小组负责,并经过精心的策划。使

用到的技术已经不仅仅局限在测试技术领域，包括了制造、维护、系统集成等领域

有很多测试在实验室中是很难模拟的，并且许多问题只有在实际使用环境中才可能被暴露出来，外场测试是一个非常好的方法，有利于提高系统的可靠性。尤其对于可靠性要求很高的系统。

在英文中 Site Testing 还有一个意思是网站测试，主要用于验证一个网站是否符合设计规格(信息、接口和可视化设计)和可用性标准。

10.7 SQL 测试

数据库在现在的软件系统中使用得相当广泛。作为数据库语言的基础 SQL 被广泛地应用在了各种查询语句、存储过程和视图当中。SQL 测试就是要检查这些 SQL 语句的正确性。这类测试可以使用类似于单元测试的一些方法，包括等价类划分、边界值分析等。同时还需要考虑 SQL 语句的执行效率，确保 SQL 语句执行时使用了合适的索引，避免全表的扫描。

大部分数据库都可以生成并显示 SQL 语句的执行路径：在 Oracle 中用 explain plan 来生成某个 SQL 的执行路径，再用 select 语句可以从 PLAN_TABLE 中显示执行路径。在 SQL Server 中有一个更友好的工具：SQL Server Query Analyzer，可以以文本和图形两种方式来表现执行路径。执行路径描述了数据库执行某个查询时的步骤，包括使用了哪些索引(包括单列索引、复合索引和聚簇索引等)、时间开销等。通过分析执行路径，可以比较容易地看出 SQL 语句执行时是否使用了合适的索引，是否进行了全表扫描以及时间开销等信息。如果查询效率较低，可以对 SQL 语句进行优化或建立更恰当的索引，接着继续分析直到查询性能最佳。一般说来，如果查询条件使用了不等于、Like 匹配，例如：name != 'tom'、name like 'tom%' 及对查询字段使用了函数 upper(name) = 'TOM'，将导致全表扫描，一定要检查。

对于商用的数据库系统，要达到一个好的测试覆盖率是非常困难的。因为其输入域的组合空间是相当庞大的。因此，这类测试一般需要借助一些测试自动化工具。你可以开发用于特定需要的工具，也可以借助商用工具，例如：RAGS 等^[270]。

10.8 2000 年测试

这已经是一个过时的测试了，一般对于新的软件和硬件系统都不会存在该问题。然而，如果需要使用到一些老的硬件和软件，就需要做这方面的测试，以验证 2000 年时间兼容性方面的问题。思科给出了一个测试千年问题的模板^[269]，具体如表 10-2 所示。

表 10-2 千年测试

测试类型	原因	特殊数据/测试
交叉测试	确定对于高风险日期的交叉工作正确性。高风险日期包括那些世纪变换的点和闰年	1999—2000 年的变化: <ul style="list-style-type: none"> • 12/31/1999 to 1/1/2000 • 12/30/1999 to 1/1/2000 • 12/31/2000 to 1/1/2001 2000 年闰年测试: <ul style="list-style-type: none"> • 2/28/2000 to 2/29/2000 • 2/29/2000 to 3/1/2000
日期有效性测试	确定输入的有效日期能像期望的一样工作。一些代码,类似 9/9/99 是被作为溢出数据对待,认为是无限的情况。设置这些特定的值,测试被测产品的期望行为	闰年日期: <ul style="list-style-type: none"> • 2/29/2000 • 2/29/2004 “无限日期”: <ul style="list-style-type: none"> • 9/9/99 • 9/10/99 • Any Random 1999 date
无效日期输入	保证无效日期输入的时候返回一个错误。一个闰年时有效的日期在非闰年时可能出错	<ul style="list-style-type: none"> • 2/29/2002 • 2/29/1997
代码检查	检查可能出现千年问题的代码区域	
实时时钟产品测试	测试实时时钟产品在 2000 年滚动期间掉电时的行为。这类测试仅应用于那些有备份电池时钟的产品	1999 年向 2000 年滚动时掉电; 滚动后掉电
时区测试	对于有时区特性的产品,测试 1999 年—2000 年滚动情况	时区向后测试; 时区向前测试
年范围检查	检查“报告/输出/日志”中的年份值是否正确	
输入文件检查	确定自动化过程是否正确更新数据文件中的年份域	
日期/天的周映射检查	检查周的日期/天映射是否正确	
时间协议测试	检查产品接受来自 NTP 的时间更新能力和过程。检查产品接受到不正确日期/时间更新的处理能力	
年 0 的映射	检查产品在输入 0 的时候是否映射到 2000 年	

10.9 回归测试

一个系统在其演进的过程中面临着不断的变更,这些变更一方面来自修改不断发现的缺陷,另一方面来自新特性的增加。无论是哪种原因造成的变更,我们都将面临着一个令人烦恼的话题——测试的回归(Regression Testing)。回归测试最主要的目的是验证对系统的变更没有影响以前的功能,并且保证当前功能的变更是正确的。回归测试令人烦恼的最主要原因在于频繁的重复性劳动。一个再有耐心的测试人员在重复做同一个测试超过 5 遍或 6 遍时就会变得歇斯底里。况且有时候可能需要回归 10 遍以上。

回归测试可以发生在任何一个阶段,包括单元测试、集成测试和系统测试。回归测试的策略需要尽早确定,而不是到了需要发生时才决定。在确定回归测试策略时,主要需考虑两个方面。

- 回归测试的范围

在回归测试范围选择上,一个最简单的方法是每次回归执行所有在前期测试阶段建立的测试来确认问题修改的正确性以及没有造成对其他功能的不利影响。很显然,这种回归的成本是高昂的。

另外一种方法是有选择地执行以前的测试用例。这时,回归的时候仅执行先前测试用例的一个子集,因此子集选取是否合理,是否具有代表性将直接影响回归测试的效果和效率。常用的用例选择方法可以分为3种:

(1) 局限在修改范围内的测试。即这类回归测试仅根据修改的内容来选择测试用例,这部分测试用例仅保证修改的缺陷或新增的功能被实现了。这种方法的效率是最高的,然而风险也是最大的,因为它无法保证这个修改是否影响了别的功能。该方法在进度压力很大,或者系统结构设计耦合性很小的状态下可以被使用。

(2) 在受影响的功能范围内回归。这类回归测试需要分析当前的修改可能影响到哪部分代码或功能,对于所有受影响的功能和代码,其对应的所有测试用例都将被回归。如何判断哪些功能或代码受影响依赖于开发过程的规范性和测试分析人员(或开发人员)的经验。对于开发过程有详细的需求跟踪矩阵的项目而言,在矩阵中分析修改功能所波及的代码区域或其他功能是比较简单的,同时有经验的开发人员和测试人员能够有效地找出受影响的功能或代码。对于单元测试而言,代码修改的影响范围需要充分考虑到一些对公共接口的影响,例如:全局变量、输入输出接口变动、配置文件等。该方法是业界推荐的方法,适合于一般项目的使用。

(3) 根据一定的覆盖率指标选择回归测试。该方法一般是在相关功能影响范围难以界定的时候使用。最简单的策略是规定修改范围内的测试是100%。其他范围内的测试规定一个用例覆盖阈值,例如:60%。

- 回归测试的自动化

我们在前面讲到测试用例设计时,一直都在强调,用例的设计要利于回归。不利于回归的测试用例将给回归工作带来很大的压力。众所周知,回归测试是繁琐的、累人的工作,因此提高回归测试自动化工作可以把测试人员从重复的工作中解脱出来。

回归测试的自动化包括测试程序的自动运行、自动配置,测试用例的管理和自动输入,测试信息与结果的自动采集,测试结果的自动比较和结论的自动输出。对系统测试功能比较简单、测试界面相对稳定并且测试用例良好组织的测试来说,采用“捕获/回放”工具是比较合适的,这类工具有 WinRunner、QARun、Visual Test 等。对于复杂的测试来讲,探索开发通用的或者专用的自动化测试工具是灵活且易于扩充的方法。一个实现良好的自动化测试程序包含上面提到的自动化基本元素是必需的,尤其测试用例的组织 and 输入、结果的收集与判断是关键中的关键,是需要仔细考虑的问题。这类工具一般都要使用到脚本语言,业界比较流行的脚本语言包括: TCL、Python、Expect、Perl、YACC、GCC 等。

10.10 本章小结

本章介绍了一些特定的测试方法。这些方法可以作为前几章的补充或延伸。这里，并没有详细说明这些测试如何使用，仅做了一些简单的介绍。读者若感兴趣，可以参考相关文献。

第 11 章 软件质量透视

通过前面各章的学习，对传统的软件测试有了一个比较全面的认识。我们经常说测试是软件质量的捍卫者，是产品最终进入到用户之前的最后一道防线。尽管这么说给测试人员无形中增加了很多不应有的压力，但有一点是明确的，即测试人员需要对产品的质量负责。那么什么是质量，该如何来看待质量呢？本章将全面解释软件的质量，通过本章学习，应当掌握：

1. 什么是质量？
2. 如何才能提高软件产品的质量？
3. Deming 的 14 条质量原则是什么？
4. CMM、PSP 和 TSP 之间的关系是什么？

11.1 质量的定义

作为软件产品的销售人员、市场人员或维护人员经常会受到客户这样那样的指责或抱怨，客户说：你们产品的质量太差，不稳定等等。那么什么是质量，我们该如何来衡量质量呢？

在韦氏字典中，质量被定义为“一些东西的本质特征，一种内在的特征，代表着优秀的程度和等级”。如果你看过计算机方面的文章，你将发现质量的定义略有不同。

ISO 关于质量的定义表示如下：

“一个实体(产品或服务)的所有特性，基于这些特性可以满足明显的或隐含的需要。”从这个定义，我们可以引申出质量的 3 个维度：

- 符合目标。目标是客户所定义的，符合目标即判断我们是不是在做我们需要做的事情。在这个维度上用户认为的质量是“产品是否按用户的需要运行”。换句话说就是产品是否适合使用。这也应当是产品目的的一种描述。比较典型的一个例子是记录在用户的需求规格说明书中，并且质量系统始终围绕着需求进行质量的改进和质量的检测。
- 符合需求。即产品是否在做让它做的事情。如果想要一个有质量的产品，需求必须是可度量的，并且产品的需求要么被满足，要么不被满足。根据这种度量，质量是一个二维的状态，也就是说产品要么是有质量的，要么是没有质量的。需求可能非常复杂或者非常简单。但只要它们是可以度量的，它们就可以被用来确定质量是否被达到或没有达到。这是生产者关于质量的观念，他们认为质量就是满足生产者的需求或规格说明书。在此满足规格说明书成为了产品本身的一个终点。
- 符合实际需求。实际的需求包括用户明确说明的和隐含的需求。而往往我们会忽

略隐含的需求。因此在控制一个产品的质量的过程中必须关注这些隐含的需求，并给予应有的验证。

产品是为客户提供服务的，凡是不满足客户需求的，我们都认为是一个失效(failure)。所以我们的产品始终必须围绕着客户的需求进行开发和验证。

在这里，我们谈到了客户。其实在一个软件需求的收集过程中需要关注客户和用户。而我们常常会忽略客户与用户之间的区别。那么谁是客户，谁是用户呢？简单地说，客户是真正能够决定是否购买你软件的人，而用户是实际使用软件的人。了解了这个区别，在分析需求的重要性时就可以进行参考。同时在产品质量验证时也可以做出不同的权衡。另一方面，在考虑用户需求时，往往只考虑实际使用软件的人员，而忽略了其他一些人员对软件的要求或对软件造成的潜在竞争，这包括维护人员的要求、系统管理人员的要求、软件上下游人员的要求、先前版本的情况、市场上竞争对手的软件情况等。

当每个人提到质量时，经常会遇到下列矛盾，在这些矛盾中隐含着对质量的承诺^[8]：

- 质量需要一个承诺，尤其是高层管理者的承诺。但为了得到质量，高层管理者必须和其雇用的员工进行紧密合作；
- 许多人相信没有缺陷的产品和服务是不可能的。但是控制在一定级别的缺陷数是正常并可接受的；
- 质量经常是和成本紧密联系在一起，一个高质量的产品同时也意味着高投入。这是设计的质量和一致性质量的一个矛盾；
- 一个高的质量要求需求规格说明书足够详细，并根据这些规格说明书产品可进行定量的分析。然而许多组织没有能力或者不愿意制作如此详细的规格说明书；
- 技术人员经常相信规范和标准会束缚他们的创造力，因此就不遵照标准做事。然而如果要得到高质量的产品，就必须遵循良好定义的标准和过程。

11.2 质量的预防和检测

质量是不能够通过评价一个已经完成的产品而得到的，因为在产品设计之初质量已经被隐含了。因此为了得到好的质量，必须在最初的时候就开始防止质量缺陷和不足，并且通过质量保证方法来使产品变得可以评价。一些质量保证方法包括：通过软件开发过程标准结构化软件开发行为，并且用方法、技术和工具来支持软件开发过程。

此外，对质量管理程序来说，一个产品的评价和过程的评价是非常基本的。例如，包括编程规范文档，标准、方法和工具的规定及使用，数据备份过程，变更管理过程，缺陷文档化过程和缺陷恢复过程。

质量管理可以降低产品的成本，这是因为缺陷发现和修改得越早，所花费的成本就越少。尽管在初期时，其成本投入是相当大的，但是从长远来看，质量管理将减少后期的维护费用，并最终得到一个高质量的产品。研究表明：维护阶段发现和修改一个缺陷的成本是需求阶段的70倍甚至更高(参考第1章第1节的相关内容)。

一个有效的质量管理成本包括4个部分——预防成本、检视成本、内部缺陷发现和修改成本、外部缺陷发现和修改成本。预防成本包括防止缺陷最初产生的活动的成本。检视

成本包括测量、评价和审计一个产品或服务是否和标准与规格相一致而花费的成本。内部缺陷发现和修改成本是那些在产品交付之前发现和修改错误所投入的成本。外部缺陷发现和修改成本是产品发布之后发现和修改缺陷所花费的成本。外部缺陷有可能是灾难性的，因为它们可能会损害企业的名誉并最终导致销售的下滑。

预防最大的回报是在增加预防成本的同时减少外部缺陷，进而提高产品质量，减少维护成本和产品成本。

11.3 如何提高软件产品的质量

软件质量是每个人都想要的东西。经理们知道他们想要高质量的产品。软件开发人员知道他们想要产生高质量的产品。用户坚持软件必须可靠、持续地工作。有许多组织的软件质量保证组期望提高和评价他们的软件应用。然而对质量保证来说没有一个可接受的实践。这样不同组织的质量保证组可以使用不同的过程、扮演不同的角色及执行他们的计划。

如何提高软件的质量已经不是一个纯粹的技术问题，而是一个工程问题。自从软件危机产生以来，出现了很多关于提高产品质量方面的理论和方法，有从技术角度出发的，例如：面向对象技术的产生和推广，第四代语言的诞生等；也有从自动化工具入手的，例如：CASE 工具、过程控制软件、自动化管理平台等；也有从过程模型角度出发的，例如：迭代模型、螺旋模型、RUP、IPD、净室软件工程等；也有从管理角度出发的，例如：团队管理、绩效管理、PSP、TSP 等；也有从测试角度出发的，例如：把 Deming 的 PDCA 循环应用到全流程的测试过程上等；一些相应的规范和标准也孕育而生，例如：ISO9000 系列、CMM、QMS 等。然而每一种技术都不是绝对的，软件质量的提高应该是一个综合的因素，需要从各个方面进行改进，同时还需要兼顾成本和进度。

11.3.1 流程对质量的贡献

从一个软件企业的长远发展来看，如果要提高产品的质量首先应当从流程抓起，规范软件产品的开发过程。这是一个软件企业从小作坊的生产方式向集成化规范化的大公司迈进的必经之路，也是从根本上解决质量问题，提高工作效率的一个关键手段。

软件产品的开发同其他产品（如汽车）的生产有着共同特性，即需要按一定的过程来进行生产。在工业界，流水线生产方式被证明是一种高效的，且能够比较稳定地保证产品质量的一种方式。通过这种方式，不同的人员被安排在流程的不同位置，最终为着一个目标共同努力，这样可以防止人员工作间的内耗，极大地提高了工作效率。并且由于其过程来源于成功的实例，因此其最终的产品质量能够满足过程所设定的范围。软件工程在软件的发展过程中吸取了这个经验并把它应用到了软件开发中，这就形成了软件工程过程，简单地说就是开发流程。

不管我们做哪件事情，都有一个循序渐进的过程，从计划到策略到实现。软件流程就是按照这种思维来定义我们的开发过程，它根据不同的产品特点和以往的成功经验，定义

了从需求到最终产品交付的一整套流程。流程告诉我们该怎么一步一步去实现产品,可能会有哪些风险,如何去避免风险等。由于流程来源于成功的经验,因此,按照流程进行开发可以使我们少走弯路,并有效地提高产品质量,提高用户的满意度。

目前流行的流程方法有很多种,不同的过程模型适合于不同类型的项目。瀑布模型是应用最为广泛的一种模型,也是最容易理解和掌握的模型,然而它的缺陷也是显而易见的。遗漏的需求或者不断变更的需求会使得该模型无所适从。然而,对于那些容易理解但很复杂的项目,采用瀑布模型会比较适合,因为这样可以按部就班地去处理复杂的问题。在质量要求高于成本和进度时,该模型表现得尤其突出。

螺旋模型也是一个经典模型,它关注于发现和降低项目的风险^[20]。螺旋型项目从小的规模开始,探测风险,制定风险控制计划,接着确定下一步项目是否还要继续,然后进行下一个螺旋的反复。该模型的最大优点就是随着成本的增加,风险程度随之降低。然而螺旋模型的缺点是比较复杂,且需要管理人员有责任心,专注以及有管理经验。

RUP(Rational Unified Process)是Rational公司提出的一套开发过程模型。它是一个面向对象软件工程的通用业务流程^[20]。它描述了一系列相关的软件工作流程,它们具有相同的结构,即相同的流程构架。RUP为在开发组织中分配任务和职责提供了一种规范方法,其目标是确保在可预计的时间安排和预算内开发出满足最终用户需求的高品质的软件。RUP具有两个轴,一个轴是时间轴,这是动态的。另一个轴是工作流轴,这是静态的。在时间轴上,RUP划分了4个阶段:初始阶段、细化阶段、构造阶段和发布阶段。每个阶段都使用了迭代的概念。在工作流轴上,RUP设计了6个核心工作流程和3个核心支撑工作流程。核心工作流程包括:业务建模 workflow,需求 workflow,分析设计 workflow、实现 workflow、测试 workflow和发布 workflow。核心支撑 workflow包括:环境 workflow、项目管理工作流和配置与变更管理工作流。具体可以参考图 11-1。RUP 汇集现代软件开发中多方面的最

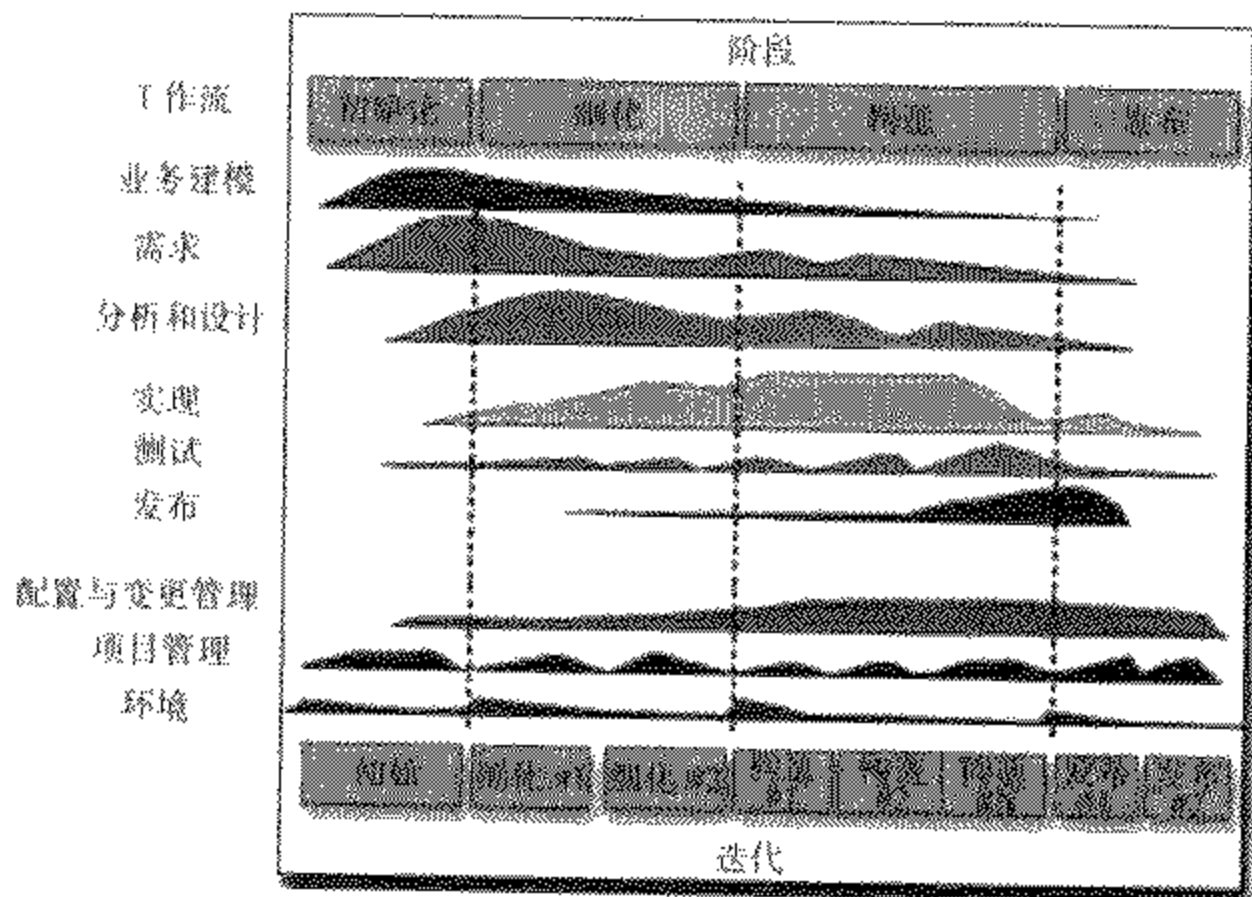


图 11-1 RUP 流程示意图

佳经验，并为适应各种项目及组织的需要提供了灵活的形式。作为一个商业模型，它具有非常详细的过程指导和模板。但是同样由于该模型比较复杂，因此在模型的掌握上需要花费较大的成本。尤其对项目管理者提出了更高的要求。

IPD(Integrated Product Development)流程是由IBM提出的一套集成产品开发流程，非常适合于复杂的大型开发项目，尤其涉及到软硬件结合的项目。IPD从整个产品角度出发，流程综合考虑了从系统工程、研发(硬件、软件、结构工业设计、测试、资料开发等)、制造、财务到市场、采购、技术支援等所有流程，是一个端到端的流程。在IPD流程中总共划分了6个阶段(概念阶段、计划阶段、开发阶段、验证阶段、发布阶段和生命周期阶段)，4个决策评审点(概念阶段决策评审点、计划阶段决策评审点、可获得性决策评审点和生命周期终止决策评审点)以及6个技术评审点，具体可以参考图11-2。IPD流程是一个阶段性模型，具有瀑布模型的影子。该模型通过使用全面且复杂的流程来把一个庞大而又复杂的系统进行分解并降低风险。一定程度上，该模型是通过流程成本来提高整个产品的质量并获得市场的占有。由于该流程没有定义如何进行流程回退的机制，因此对于需求经常变动的项目该流程就显得不太适合了。并且对于一些小的项目，也不是非常适合使用该流程。

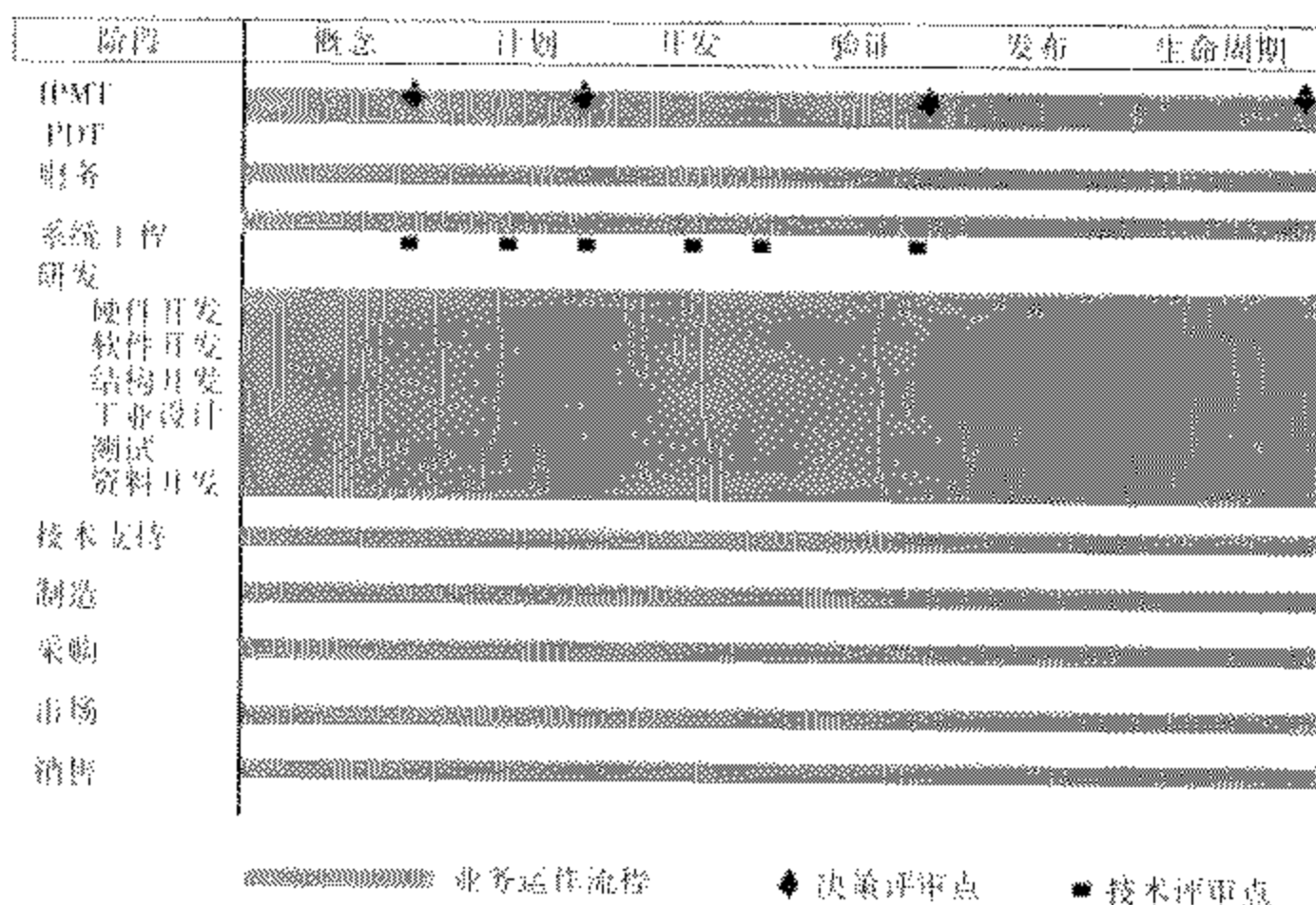


图 11-2 IPD 流程示意图

11.3.2 流程与技术

流程 and 成功不是等价的。没有流程就，成功就不可能得到保证，但有了流程并不意味着肯定能够成功。这恐怕是很多迷信于流程的人所不能接受的。但这的确是事实。记得有个做了将近30多年的需求分析专家说过：即使是一个已经达到CMM4级的公司，也完全有可能做不好需求分析。为什么？技术。技术是成功的另外一个必要条件。就好比现在你要从上海到北京去，流程给你指出了最短的路径，技术提供给你最快的交通工具，两者结

合就是完美。

对于软件开发来说,要保证软件的质量,需要掌握多方面的技术,包括分析技术、设计技术、编码技术和测试技术等。在国内有一个普遍的非正常现象,就是大家觉得只有编程才是玩电脑的真正技能。就好像造一套房子,其他都不重要,只要砖瓦匠有高超的技能就行了。尽管这个比喻会打击很多程序员的自尊心,但这的确是一个事实。我们缺少系统级的工程师,在分析和设计方面的工作做得很不扎实。

需求是一个项目的灵魂。模棱两可的需求带来不可避免的后果便是返工——重做一些你认为已做好的事情。返工会耗费开发总费用的 40%,而 70%~85% 的重做是由于需求方面的错误所导致的^[208]。想象一下如果你能减少一半的返工会是怎样的情况?你能更快地开发出产品,在同样的时间内开发更多、更好的产品,甚至能偶尔回家休息休息。在《软件需求》^[207]一书中关于如何进行需求分析给出了比较详细的介绍,RUP 中关于需求的指导也是很实用的。

设计是最能体现一个工程师能力的地方。一个好的设计基本上决定了产品的最终质量。设计是把需求转换成系统的一个关键步骤,它需要从自然语言描述的需求中寻找出设计的基础单元,构建出整个系统的构架。在 RUP 中关于系统构架师和设计师的定位是相当高的。关于设计方面的技能是很广的,包括从传统的结构化设计到面向对象设计。设计人员需要掌握一定的建模技术。UML 是国际上比较流行的一种建模语言^[210]。在嵌入式方面,SDL 也是一种非常好的选择。《设计模式》^[206]是一本在设计思想方面总结得非常出色的书,作为一名设计人员(尤其是面向对象设计人员)必须好好研究一下。但是对这些模式的应用应当讲究一种自然的应用,千万不要因为模式而去设计模式,否则会适得其反。

现在的程序员热衷于掌握多种编程语言,或者过分讲究语言的技巧化,而往往忽略了编程语言的规范化。不规范的语言应用给程序的可理解性、可维护性以及可测试性带来很大的不便,甚至会损害产品的质量。某公司曾对中国程序员和印度程序员做过一个测验,这个测验要求参加者对一组数进行排序。测试结果发现,印度程序员设计的程序使用的算法并不是最优,但却是最不容易出错的,并且几个程序员写出来的代码如出一辙。而几个中国程序员写出的代码,有的非常漂亮,很精练,效率很高;有的却很冗杂,还有错误。如果大家是在做研究性的项目或纯粹兴趣性的项目,那么大家充分发挥自己的编程天才也无可厚非。然而,对于一个软件公司,其产品最终是要交给用户的,需要遵循的是一个软件产品的开发工程。因此这类软件的开发需要遵循一定的编程规范,毕竟开发的软件不是自己用,还需要和别人集成,还需要给以后版本重用和维护。

测试的技术将在 11.3.4 节进行阐述。流程很关键,技术也很重要,总之,“鱼”和“熊掌”,两者都不能放。

11.3.3 全面质量管理

作为全面质量管理的最初倡导者和成功者 Deming,其质量理论包括统计理论的使用,他相信当一个过程出现变化时,统计是最小化矛盾的核心^[8]。统计学还帮助我们理解过程本身,对过程获取控制并改进它们。这可以引用一句话“我们相信在上帝那边,所有人都必须使用数据”。统计有助于我们特别关注于问题的主要因素,当问题改正后能极大地

提高产品的质量 Deming 指出许多统计技术是不难的, 但需要一个较强的数学基础。然而, 教育是一个强有力的工具, 并且需要一个组织的各个层次使统计产生作用。

1. Deming 的质量原则

Deming 列出了 14 条质量原则, 这些质量原则必须同时运用才有效, 下面是关于这些原则的讨论, 并对每个原则描述了质量保证组织运用这些原则的方法。

第一个原则: 要有一个坚定不移的目标

许多公司趋向于解决当前的问题而忽略将来的目标。根据 Deming 的原则: “你可以很容易地站在纠缠不清的问题的边缘, 并且在将来变得越来越有效。但是对于一个公司, 在没有一个针对未来的计划的前提下, 它是不能存在于商业领域中的。”一个坚定不移的目标需要创新。如要有一个长期的计划, 投资于研究和教育, 并且不断地改进产品和服务

为了应用该原则, 一个质量保证组织可以:

- 开发一个质量保证计划, 提供一个长期的质量方向;
- 需要软件测试者为每个项目开发并维护一个一致的测试计划;
- 鼓励质量分析人员和测试人员遵循具有革新方法来最大化产品的质量;
- 致力于不断改进质量过程。

第二个原则: 吸收新的哲学

质量必须成为一个新的信仰, 根据 Deming 的理论: “生存的成本和需要花钱购买的商品和服务是成反比的, 如: 可靠的服务可以降低成本, 延迟或错误却会提高成本。”由于延迟和错误, 商品和服务的消费被终止了, 这降低了它们存在的标准。在系统中, 对可接受的层次和缺陷的忍耐程度是质量和产量之间的一个路障。

为了应用该原则, 一个质量保证组织可以:

- 教育信息技术组织关于质量的价值和需要;
- 提高质量保证部门的地位, 使它们和别的部门同样重要;
- 纠正对质量部门是“看门狗”的消极看法。

第三个原则: 不要依赖于海量的检查

传统的想法认为检查可以排除糟糕的质量。当难于确定在过程中一个缺陷在哪边产生时, 一个好的方法是通过检查关注于我们做得如何, 而不应针对最终的产品。质量应当是内在的, 而不是依赖于无数的检查获得的。

为了使用该原则, 一个质量保证组织可以:

- 在整个开发生命周期中, 提高并使用技术评审、走读和检视来获取质量;
- 在整个组织中灌输质量意识, 并把它作为一个切实的、可度量的工作产品;
- 需要信息技术质量的统计证据。

第四原则: 结束基于纯粹价格标签裁定的商业实践

“两个或多个以上的同种商品的供应商会成倍地加剧损害, 这些损害是内在必然的, 并且对任何一个供应商都是不利的。”为了最好地服务于公司, 购买方可以通过和供应商建立一种长期的忠实的关系, 并且与一个专门的供应商建立信任关系。通过一些标准来衡量哪个供应商质量更好这不是一个好的方法, 一个更好的方法是积极地使用 Deming 的 14 条原则参与到供应商管理当中。

为了应用该原则，一个质量保证组织可以：

- 需要软件质量和测试提供者提供能够证明他们质量的统计数据；
- 对每一个质量保证工具、测试工具和服务选择一个最好的供应商，并且建立一个与质量计划一致的工作关系。

第五原则：产品和服务系统稳定地、长期地提高

改进不是一时的努力——管理者有责任不断地提高质量。“把火扑灭——很多公司称之为救火队，并不是改进。寻找失去控制的地方，找到其特殊的原因并改正它，这只是把过程纠正回本来应该的位置。改进的责任是一个无止境的过程”。

为了应用该原则，一个质量保证组织可以：

- 不断地提高质量保证和测试过程；
- 不要依赖于主观的判断；
- 使用统计技术，如测试分析和通过主因及效果分析揭示问题根源等方法。

第六原则：建立培训和再培训制度

在很多公司，通常只有很少甚至没有培训，工人们不知道何时才能正确地完成他们的工作。消除不适合的培训是非常困难的。Deming 强调：只要工作成果还无法受到统计控制，并且还能获得更大的好处时，培训就不应当被中止。

为了应用该原则，一个质量保证组织可以：

- 建立现代的培训辅助和实践；
- 鼓励质量工作者通过参加研讨班或上课不断地提高质量和测试方面的技术知识；
- 奖励工作者建立新的研讨班和特殊的兴趣小组；
- 使用统计技术确定何时培训被需要，并且何时培训可以结束。

第七原则：建立领导阶层

“没有任何一个借口可以把人放到他们不知道如何做的岗位上。绝大多数所谓的‘游手好闲的人’是因为他们被放置在不适合的工作上或者由于不善的管理造成的。”去寻找是什么原因造成工人们不以工作自豪是管理者的责任。从一个信息技术的眼光来看，开发人员经常认为质量这种事应当是质量部门的责任。QA 作为质量领导者应当敢作敢为，并且指出质量是每一个人的责任。

为了应用该原则，一个质量保证组织可以：

- 如果一个开发人员有大量的错误被 QA 测试发现，那么需要花时间来培训他（她）如何进行单元测试或如何更有效地编码；
- 提高监督，这是管理者的责任；
- 允许项目管理者有更多的时间在工作上去帮助项目组内的人；
- 使用统计技术来揭示哪里存在缺陷。

第八原则：驱除恐惧

在问题的解决上一般都缺乏创意。提议新的观点是太冒险了。人们害怕丢失加薪、升职甚至工作的机会。“恐惧造成了可怕的代价。恐惧无处不在，它剥夺了人们的骄傲，伤害并剥夺了他们为公司做贡献的机会。如果恐惧被消除，你将无法想象会发生什么事情”。一个通常的问题是被检查的恐惧。

为了应用该原则，一个质量保证组织可以：

- 提出质量是好的、应当被奖励的观点，并且鼓励任何有助于提高质量的新观点；
- 在一个结构性走读、检视或 JAD 任务之前，QA 应当保证每个人都理解执行的规则并且提供一个公正的环境；
- 有一个周期性的“质量日”，在该日任何能够提高质量的提议被大家一起共享。

第九原则：打破不同领域间的障碍

当各部门有不同的目标时，很多的问题就随之产生了。这些部门不会像一个有效的小组一样去解决问题、设定政策和定义新的方向。“人们在自己的部门内可以工作得非常好，但如果他们的目标发生冲突时，他们将对公司造成损害。此时最好成立一个联合工作组，他们对公司负责。”

为了应用该原则，一个质量保证组织可以：

- 质量保证部门和其他部门需要紧密地工作在一起。QA 应当被视为好的伙伴，他们致力于使软件产品最终成为最优秀的产品；
- 质量保证组织应当指出缺陷应当在产品最终到达用户手上之前被发现。

第十原则：排除为工作努力设置的口号、训示及目标

“口号是不会帮助任何一个人做好工作的，它们会产生挫折和怨恨。”像“零缺陷”或“在第一时间把事情做好”等口号在表面上看是好的，问题是它们被视为一种信号，即管理者不理解或不关心雇员的问题。设定了目标但不描述如何去完成该目标，在实践中是经常发生的。

为了应用该原则，一个质量保证组织可以：

- 鼓励管理人员避免使用口号；
- QA 组织应当开发和文档化标准、过程和步骤，而不是产生一些无用的口号，其他组织可以使用这些标准、过程和步骤得到最好的质量。

第十一原则：消除数字目标

“引用或者其他一些工作标准，如测量每天的工作或速度，会比其他任何特定的工作条件更会阻止质量。当工作标准被一般性使用时，它们保证了低效和高成本。”一个合适的工作标准应当定义在质量这个方面，什么是可以接受的，什么是不可以接受的？

为了应用该原则，一个质量保证组织可以：

- 不要只关注在数字上，应当小心地关注在质量标准上；
- 避免形式化地公布个人或部门的缺陷率；
- 和开发组织一起定义质量的标准和步骤，并以此提高质量；
- 当存在一些特殊问题时，QA 和开发管理者需要非正式地描述它们。

第十二原则：消除为工作质量自豪的障碍

人被认为是一种商品，需要时被使用。如果不被需要，他们会被退回到市场。管理者可以处理许多问题但经常回避人的问题。他们经常形成“质量控制圈”，但这通常是管理者假装对该问题采取措施的借口。管理者很少授予雇员任何权威，也不理睬他们的建议。

为了应用该原则，一个质量保证组织可以：

- 慢慢灌输这样一种观念，即质量是他们应当交付的并且是非常有价值的商品；
- 员工代表的责任是寻找质量并做任何可以获得质量的活动。

第十三原则：建立一个教育和再培训的有力的过程

人们必须获得新的知识和技能。教育和再培训是对人的一种投资，这是一个长期的计划。教育和再培训必须适合人们到新的工作岗位上并承担新的责任。

为了应用该原则，一个质量保证组织可以：

- 鼓励质量工作者通过参加研讨班或上课不断地提高质量和测试方面的技术知识；
- 奖励工作者建立新的研讨班和特殊的兴趣小组；
- 在新的质量技术方面对个人进行再培训。

第十四原则：采取行动完成转变

高层管理者需要推动这 13 条原则，每个雇员，包括管理者，应当有一个关于如何持续改进质量的明确方法，但该方法最初来自于高层管理者。下面讨论的是一个可以应用 Deming 14 条原则的过程。该过程应用到测试过程中就形成了全流程的测试验证和确认过程，关于这个验证和确认过程将在下一章详细讲解。

2. 常用的质量管理工具

日本在开展全面质量管理的过程中，运用运筹学或系统工程的原理和方法，提出了很多用于质量管理方面的方法，这些理论和方法已经被广泛应用到了软件测试当中。

• 因果图 (Cause-and-Effect Diagram)

因果图经常被称之为鱼骨图，该方法可以通过头脑风暴活动来定位可能影响状态的因素。这是一个用于确定一个问题的可能原因的工具。使用影响值和可能原因的关系来表示，图 11-3 是一个因果图的例子。

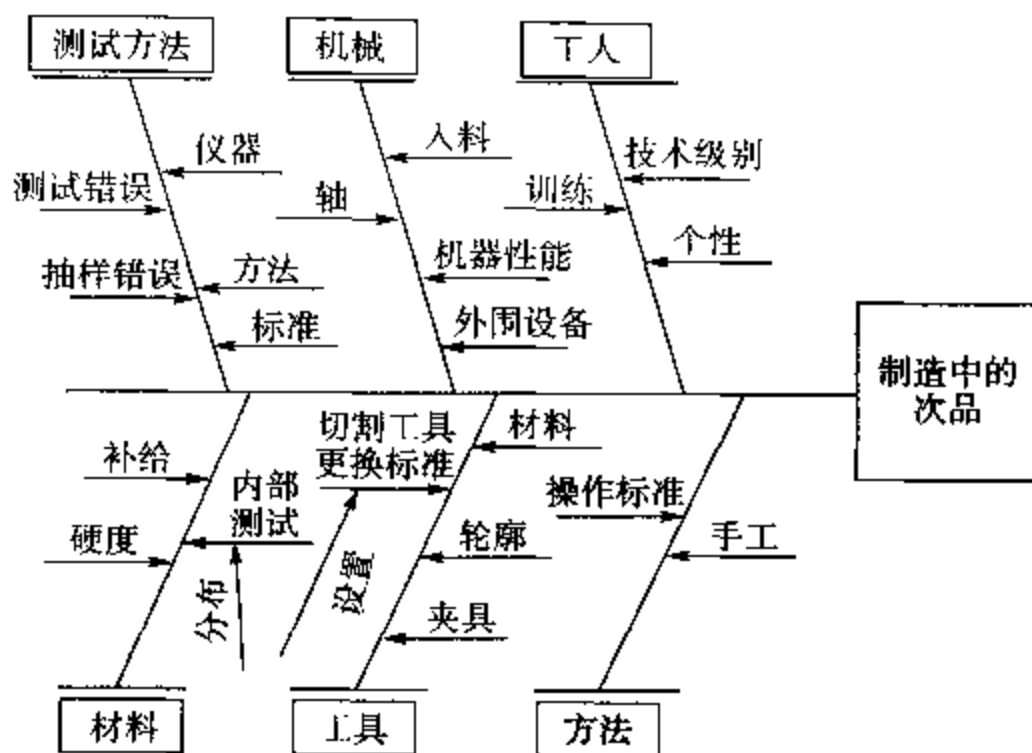


图 11-3 因果图例子

• 流程图 (Flow Chart)

这是文档化一个过程的常用的图形表示方法。该图显示用于创建一个产品或服务的一个过程或者一个工作流的顺序步骤。你首先需要了解该流程图，然后通过一定的判断来改进你的过程，图 11-4 是一个流程图的例子。

• 排列图 (Pareto Chart)

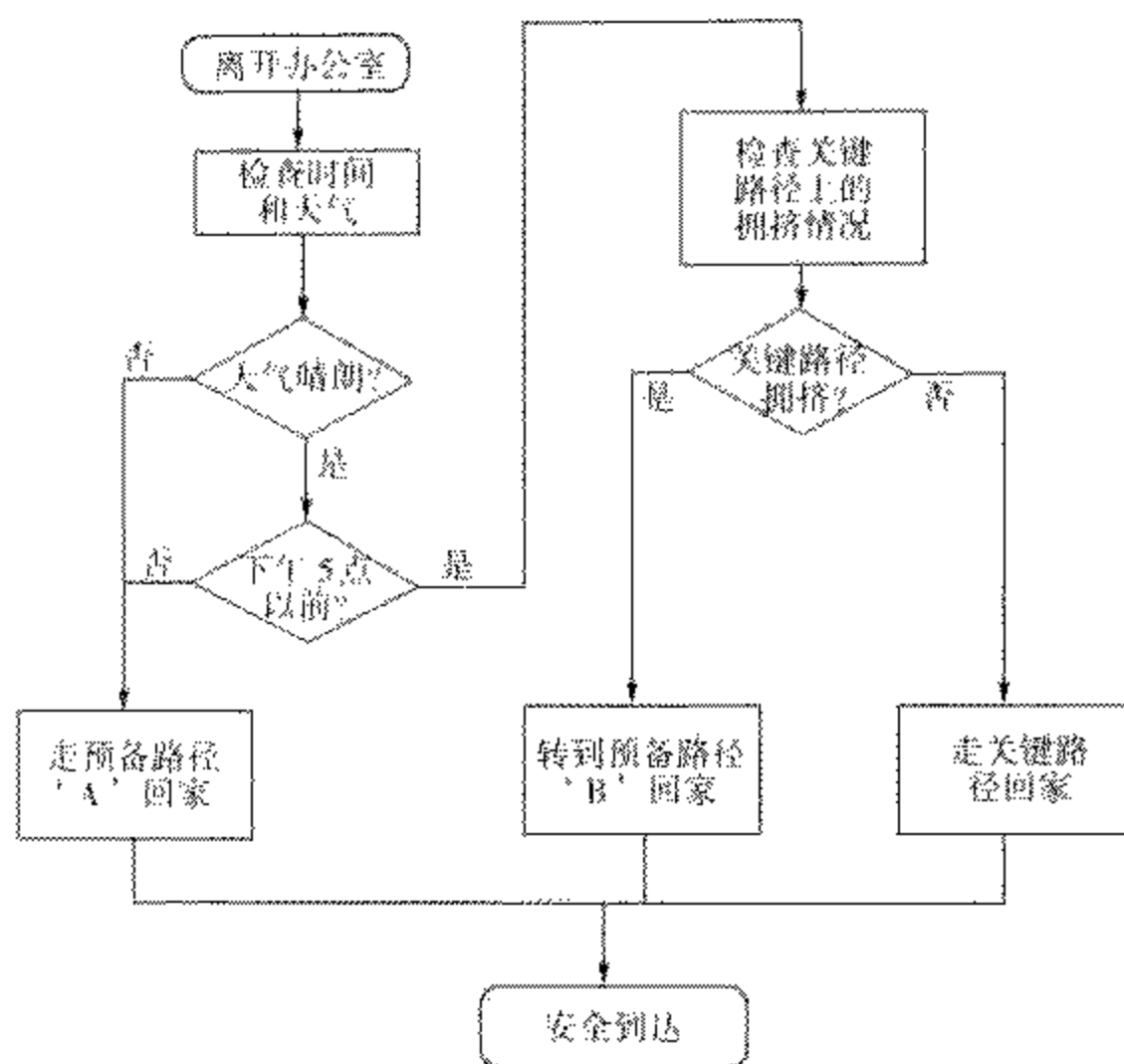


图 11-4 流程图示例

这是一个经常使用的图形技术。在该技术中，每个需要分析的事件都被命名了。意外事件通过名字进行统计。事件按照条状图中的频率，按升序进行等级划分。排列图分析使用 80/20 规则。例如一个组织的 20% 的客户占其收入的 80%，图 11-5 是一个排列图的例子。

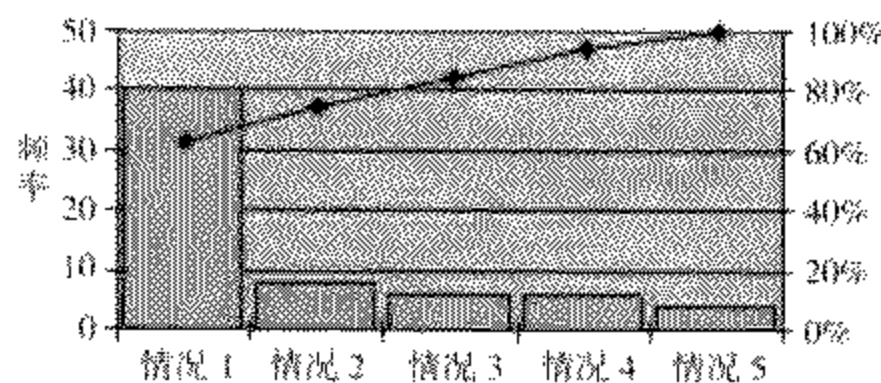


图 11-5 排列图示例

- 运行图(Run Chart)

运行图是一种图形技术，在该图形中，以时间顺序排列的数据点显示了被测特性的发展趋势，通过该图形可以发现潜在的原因而不是杂乱无章的变化，图 11-6 是一个运行图的例子。

- 柱状图(Histogram)

柱状图是对被测值的一个图形描述，这些值根据其产生的频率或相关频率表示。它同时还提供了平均值和变化值，图 11-7 是一个柱状图的例子。

- 离散图(Scatter Diagram)

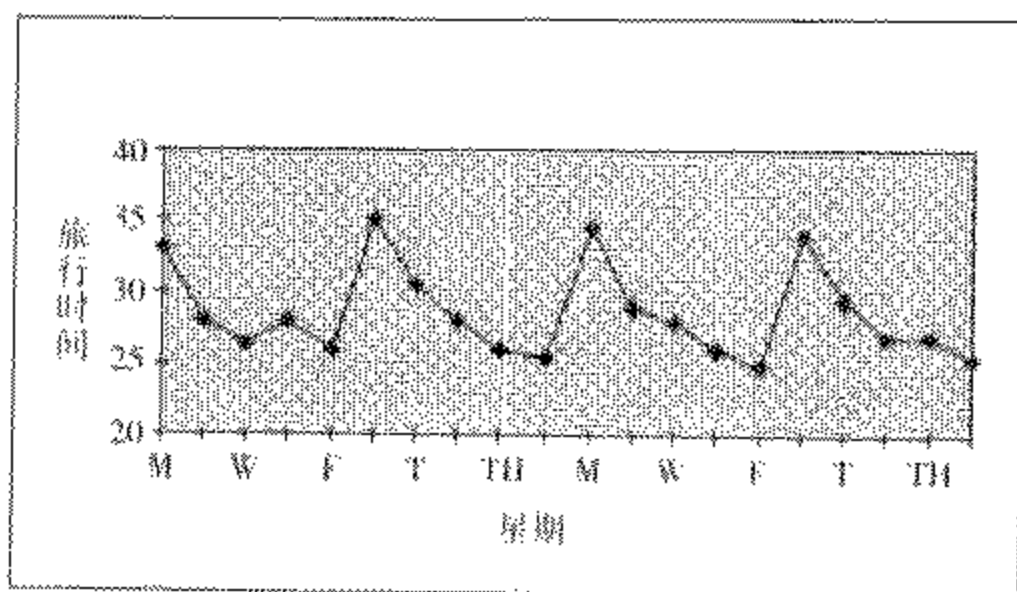


图 11-6 运行图示例

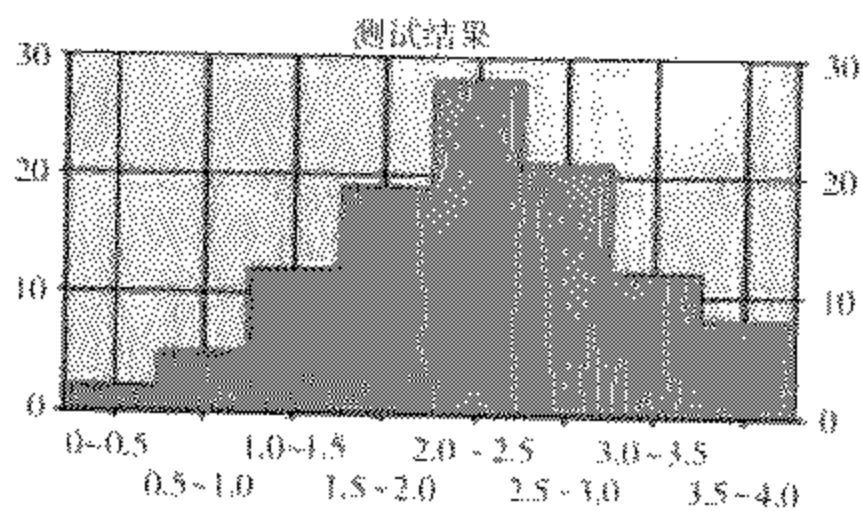


图 11-7 柱状图示例

设计离散图是用于显示两个变量或变化因子之间是否存在关系的一种图形技术。图 11-8 是一个离散图的例子。

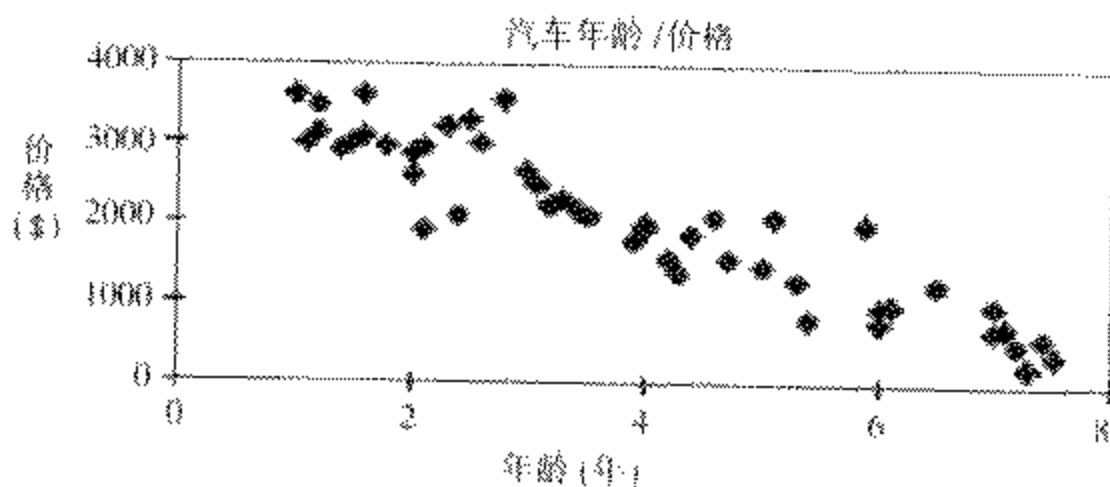


图 11-8 离散图示例

• 控制图 (Control Chart)

控制图使用一种统计的方法区分在过程中特殊和普通的变化。它是一种运行图表，根据统计的方法确定其围绕过程平均值上下的基线。图 11-9 是一个控制图的例子。

• 关联图 (Relation Chart)

关联图法，是指用连线图来表示事物相互关系的一种方法。它也叫关系图法。如图 11-10 所示，图中各种因素 A、B、C、D、E、F、G 之间有一定的因果关系。其中因素 B

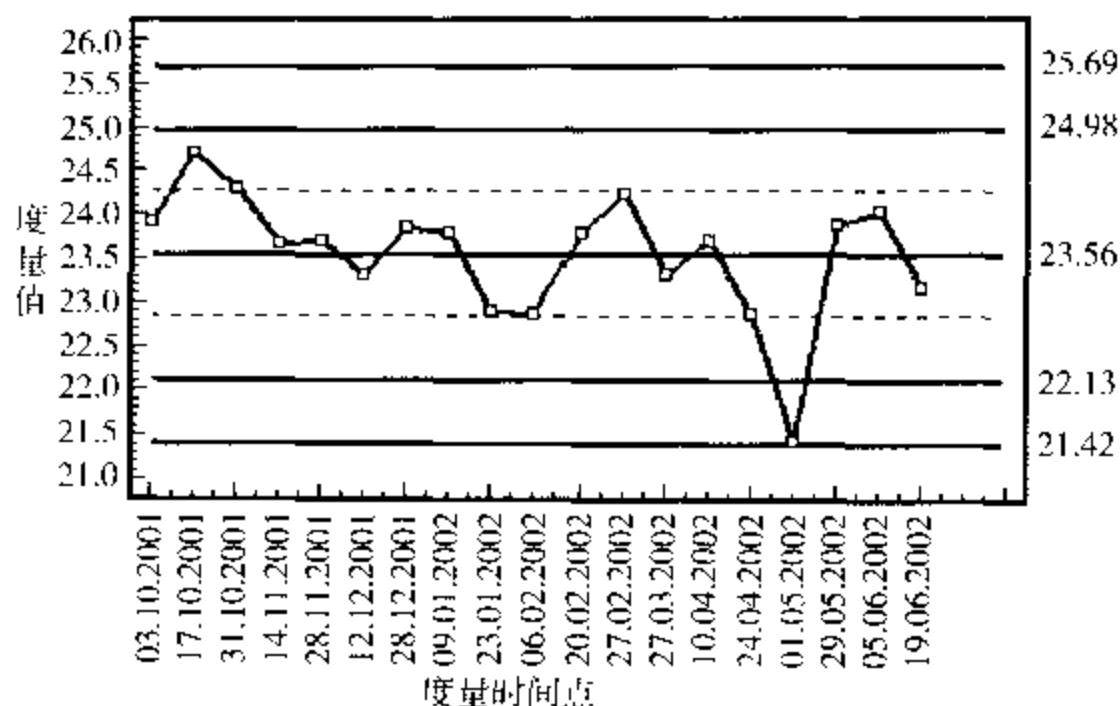


图 11-9 控制图示例

受到因素 A、C、E 的影响，它本身又影响到因素 F，而因素 F 又影响着因素 C 和 G，……这样，找出因素之间的因果关系，便于统观全局、分析研究以及拟定出解决问题的措施和计划。

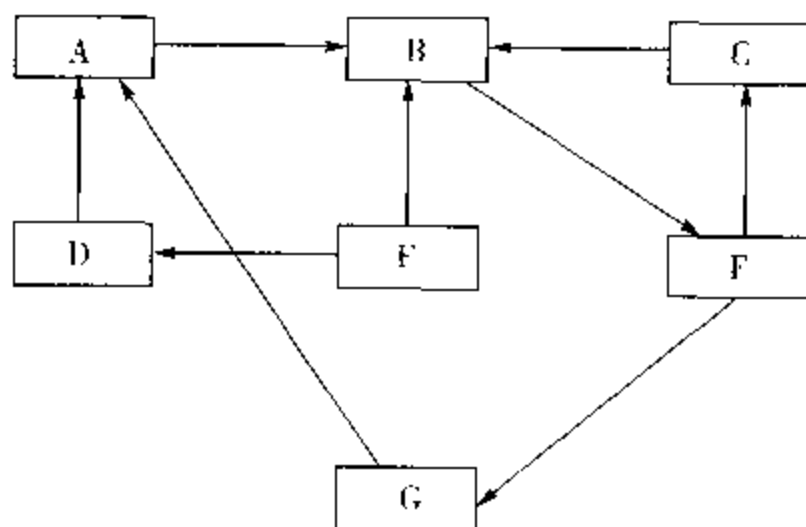


图 11-10 关联图示意

• KJ 法

KJ 法是日本川喜二郎提出的。“KJ”二字取的是川喜(KAWAJI)英文名字的第一个字母。这一方法是从错综复杂的现象中，用一定的方式来整理思路、抓住思想实质、找出解决问题新途径的方法。KJ 法不同于统计方法，统计方法强调一切用数据说话，而 KJ 法则主要靠用事实说话、靠“灵感”发现新思想、解决新问题。KJ 法认为许多新思想、新理论，往往是灵机一动、突然发现。但应指出，统计方法和 KJ 法的共同点，都是从事实出发，重视根据事实考虑问题。KJ 法的工作步骤可以表述如下：

- (1) 确定对象(或用途)
- (2) 收集语言、文字资料。
- (3) 把所有收集到的资料，包括“思想火花”，都写成卡片。

(4) 整理卡片。对于这些杂乱无章的卡片，不是按照已有的理论和分类方法来整理，而是把自己感到相似的归并在一起，逐步整理出新的思路来。

(5) 把同类的卡片集中起来, 并写出分类卡片。

(6) 根据不同的目的, 选用上述资料片段, 整理出思路, 写出文章来。

• 系统图(System Chart)

系统图法是系统地分析、探求实现目标的最好手段。在质量管理中, 为了达到某种目的, 就需要选择和考虑某一种手段; 而为了采取这一手段, 又需考虑它下一级的相应的手段。这样, 上一级手段就成为下一级手段的行动目的。如此地把要达到的目的和所需要的手段, 按照系统来展开, 按照顺序来分解, 作出图形, 以便对问题有一个全面的认识。然后, 从图形中找出问题的重点, 提出实现预定目标的最理想途径。它是系统工程理论在质量管理中的一种具体运用。图 11-11 是一个系统图的示意。

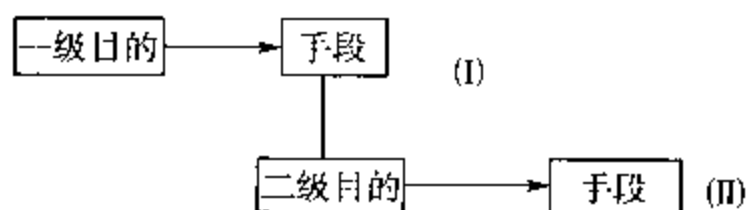


图 11-11 系统图示意

• 矩阵图(Matrix Chart)

矩阵图法是指借助数学上矩阵的形式, 把与问题有对应关系的各个因素, 列成一个矩阵图; 然后, 根据矩阵图的特点进行分析, 从中确定关键点(或着眼点)的方法。这种方法, 先把要分析问题的因素, 分为两大群(如 R 群和 L 群), 把属于因素群 R 的因素(R_1, R_2, \dots, R_m)和属于因素群 L 的因素(L_1, L_2, \dots, L_n)分别排列成行和列。在行和列的交点上表示着 R 和 L 的各因素之间的关系, 这种关系可用不同的记号予以表示(如用“○”表示有关系等)。图 11-12 为矩阵图法示意图。这种方法, 用于多因素分析时, 可做到条理清楚、重点突出。它在质量管理中, 可用于寻找新产品研制和老产品改进的着眼点, 寻找产品质量问题产生的原因等方面。

		R						
		R_1	R_2	R_3		R_4		R_m
L	L_1		○					
	L_2			⊗				
	L_3	△						
	L_i					○		
	L_n	△						

⊗密切关系 ○有关系 △像有关系

图 11-12 矩阵图法示意

• 过程决策程序图(Process Decision Program Chart)

过程决策程序图是在制订达到研制目标的计划阶段, 对计划执行过程中可能出现的各种障碍及结果做出预测, 并相应地提出多种应变计划的一种方法。这样, 在计划执行过程

中,遇到不利情况时,仍能有条不紊地按第二、第三或其他计划方案进行,以便达到预定的计划目标。它不是走着看,而是事先预计好。如图 11-13 所示,假定 A_0 表示不合格品率较高,计划通过采取种种措施(A_1 、 A_2 、 $A_3 \cdots A_p$),要把不合格品率降低到 Z 水平。

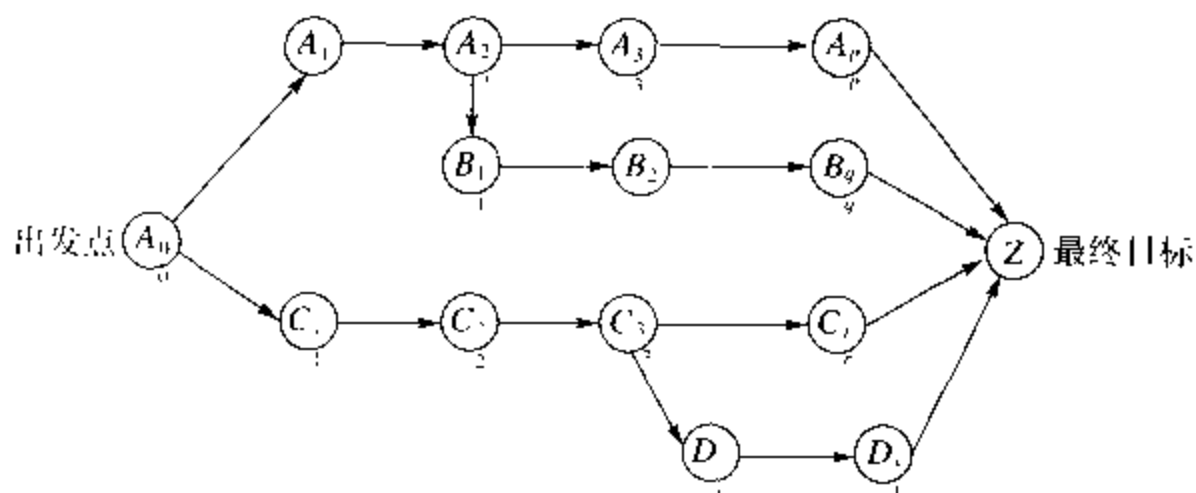


图 11-13 PDPC 示意

• 箭头图 (Arrow Chart)

箭头图法,又称矢线图法。它是计划评审法在质量管理中的具体运用,使质量管理的计划安排具有时间进度内容的一种方法。它有利于从全局出发、统筹安排、抓住关键线路,集中力量,按时和提前完成计划。图 11-14 是一个箭头图的示意。

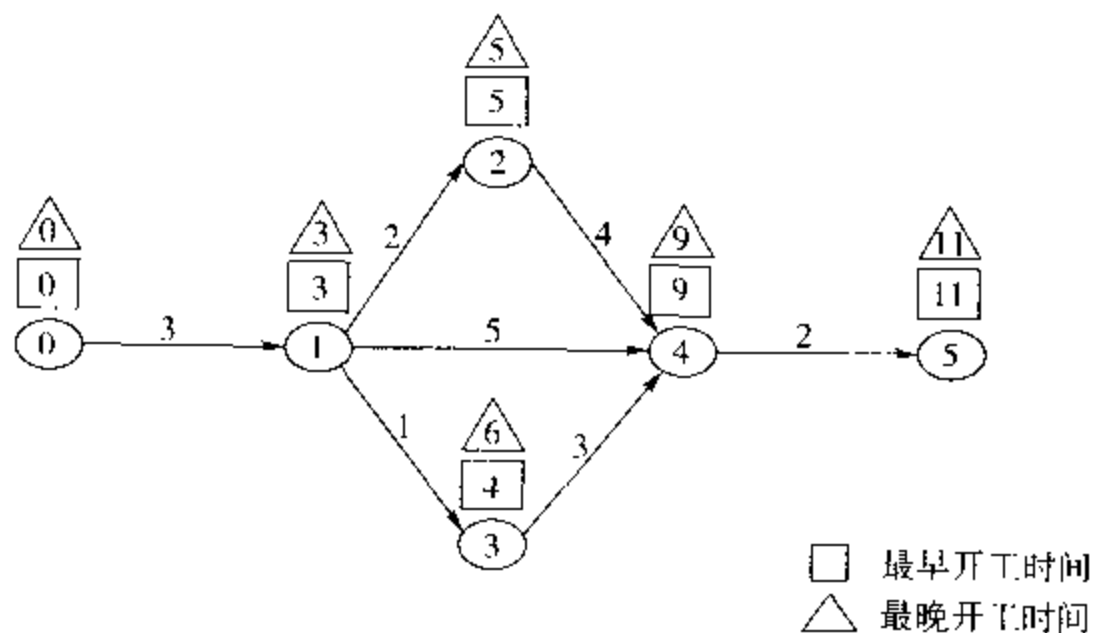


图 11-14 箭头图示意

3. PDCA 循环

“控制”这个词有多种含义,包括监视、管理、调节和约束等。在质量控制中的控制表示定义工作的目标,开发和执行一个计划以满足目标,并且检查以决定是否期望的结果产生了。如果期望的结果没有产生,那么为了完成计划,需要在工作过程中做出修改。

上面的这些可以用“Deming 循环”来描述(或 PDCA 循环),见 8.3 节。

11.3.4 关注测试

软件测试是软件质量控制中的关键活动。业界的统计数据表明,测试的成本大约占软

件开发总成本的 50%。软件测试的目的是要发现软件中的错误。一个好的测试是发现至今没有被发现的错误。传统的软件测试专注于动态测试范畴,如:单元测试、集成测试和系统测试。随着 Deming 的质量原则在测试中的深入应用,测试已经发展到了全流程的测试,包括开发过程前期的静态测试,如需求测试和设计测试,应用的方法包括了同行评审等手段。有关这部分的详细内容将在本书后面两章进行详细描述。

软件测试是产品最终进入到用户之前的最后一道防线,有着举足轻重的地位。然而,要做好软件测试却是不容易的,一方面需要同时掌握软件开发的技能和软件测试方面的技能;另一方面产品必须给予测试充分的独立性和资源保证。

11.3.5 组织、流程和人

在一个软件企业中,要能够良性地发展,必须关注组织、流程和人三者之间的关系。组织是流程成功实施的保障,好的组织结构能够有效地促进流程的实施;流程对于产品的成功有着关键的作用,一个适合于组织特点和产品特点的流程能够极大地提高产品开发的效率和产品质量,反之则会拖延产品开发进度,并且质量也无法得到保证;对企业来说,人是最宝贵的财富,他们是技术的载体,然而在组织中优秀的员工总是缺乏的,即使你拥有他们,也会有种种限制妨碍他们发挥作用。如果他们一周工作 50~60 小时,你不可能指望他们还能处理太多的挑战性的问题^[142]。对于一个软件公司来说,无论是开发人员还是测试人员,都非常关心其今后的发展通道,如果有一条清晰的技术发展线为其指明今后的发展方向的话,那么可以大大激励员工的士气和工作积极性,同时还可以为企业积累各方面优秀的人才。另外技术发展的方向应该与现在的开发流程和规范相结合,这样才有利于专业技能的提高。

总之,组织、流程和人这三者是一个企业成功的铁三角,理想的情况下他们彼此促进,糟糕的情况下他们彼此制约。

11.4 质量标准

最早进入国内的质量标准是 ISO 系列。在软件方面主要使用 ISO9000 系列标准。ISO9000 是一个非常完整的标准,并且定义了供应商设计和交付一个有质量产品的能力所需要的所有元素。ISO9002 涵盖了对供应商控制设计和开发活动所认为重要的质量标准。ISO9003 用于证明供应商在检视和测试期间检测和控制产品不一致性的能力。ISO9004 描述和 ISO9001、ISO9002 和 ISO9003 相关的质量标准,并提供了一个完整的质量检查表。

软件能力成熟度模型是目前国内软件企业中非常受欢迎的一个质量标准。并且该标准已经成为业界一个事实上的标准。CMM 为软件组织提供了一个指导性的管理框架^[142]。在这个框架的指导下:

- 软件组织可以对其软件开发、维护过程获得控制;
- 软件组织可以推进其软件工程更为文明(Culture)、推进软件过程管理更为卓越;
- CMM 通过确定当前软件过程管理的成熟度,通过标识软件的质量和过程改进中

关键的、要害的问题(Issues)，可以指导软件组织选择正确的软件过程改进策略；

- CMM 将其焦点，聚焦在一系列具体的软件过程活动上，并以侵略方式(Aggressively)达到这些活动。一个软件组织就可以稳定地、持续地改进其整个软件组织过程，使得其软件过程管理能力取得持续的、持久的、不断的增长提高。

在 CMM 中，把软件工厂分为 5 个等级：初始级、可重复级、已定义级、管理级和优化级。它们的关系可以用图 11-15 来表示。

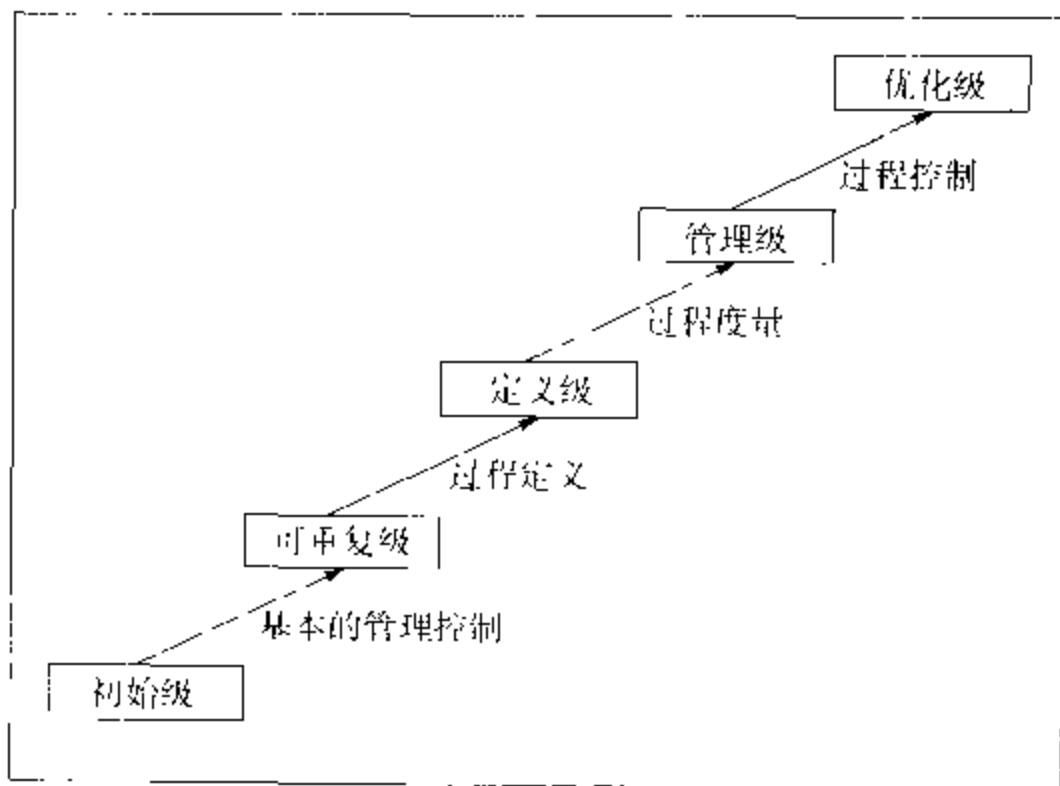


图 11-15 CMM 过程成熟度级别

其中：

- 在初始级，软件过程是未加定义的随意过程，项目的执行是随意甚至是混乱的。也许，有些企业制定了一些软件工程规范，但若这些规范未能覆盖基本的关键过程要求，且执行没有政策、资源等方面的保证时，那么它仍然被视为初始级。
- 在可重复级，人们根据多年的经验和教训，总结出软件开发的首要问题不是技术问题而是管理问题。因此，第二级的焦点集中在软件管理过程上。一个可管理的过程则是一个可重复的过程，只有过程可以重复，软件管理才能逐渐改进和成熟。可重复级的管理过程包括了需求管理、项目管理、质量管理、配置管理和子合同管理 5 个方面；其中项目管理过程又分为计划过程和跟踪与监控过程。通过实施这些过程，从管理角度可以看到一个按计划执行的且阶段可控的软件开发过程。
- 在已定义级要求制定企业范围的工程化标准，并将这些标准集成到企业软件开发标准过程中去。所有开发的项目需根据这个标准过程，裁剪出与项目适宜的过程，并且按照过程执行。过程的裁剪不是随意的，在使用前必须经过企业有关人员的批准。
- 在管理级，所有过程需建立相应的度量方式，所有产品的质量(包括工作产品和提交给用户的最终产品)需要有明确的度量指标。这些度量应是详尽的，且可用于理解和控制软件过程和产品。量化控制将使软件开发真正成为工业生产

活动。

- 优化级的目标是达到一个持续改善的境界。所谓持续改善是指可以根据过程执行的反馈信息来改善下一步的执行过程，即优化执行步骤。如果企业达到了第五级，就表明该企业能够根据实际的项目性质、技术等因素，不断调整软件生产过程以求达到最佳。

美国国防部规定，重要性达多少的软件应由相应级别以上的工厂承担。不同等级的软件工厂提交的软件，其软件质量也相差很大，国外的一份统计资料如表 11-1 所示。

表 11-1 CMM 级别与质量关系

每千行软件的缺陷数目	软件过程成熟度等级	软件准时提交的百分比	每人每月生产的程序行数	软件需要返工的百分比	平均软件失效时间(近似)
> 10	初始级	≤50	Z	≥45	2 ~ 60 分钟
< 10	可重复级	90	1.5Z	20	1 ~ 160 小时
< 1	已定义级	99	2.5Z	10	不确定
< 0.1	管理级	降低开发时间到 1/2	5 Z	5	不确定
< 0.01	优化级	降低开发时间到 1/4	10Z	≤2	近似完全可靠

CMM、PSP 和 TSP

对于很多已经推行或者准备推行 CMM 的公司来说，CMM 的起步是很难的，因此 Humphrey 又提出了 PSP(Person Software Process)^[122] 和 TSP(Team Software Process)^[121]。

CMM 是过程改善的第一步，它提供了评价组织的能力、识别优先改善需求和追踪改善进展的管理方式。企业只有开始 CMM 改善后，才能接受需要规划的事实，认识到质量的重要性，才能注重对员工经常进行培训，合理分配项目人员，并且建立起有效的项目小组。然而，它实现的成功与否与组织内部有关人员的积极参加和创造性活动密不可分。

PSP 能够指导软件工程师如何保证自己的工作质量，估计和规划自身的工作，度量和追踪个人的表现，管理自身的软件过程和产品质量。经过 PSP 学习和实践的正规训练，软件工程师们能够在他们参与的项目工作之中充分运用 PSP，从而有助于 CMM 目标的实现。

TSP 结合了 CMM 的管理方法和 PSP 的工程技能，通过告诉软件工程师如何将个体过程结合进小组软件过程，并将后者与组织进而整个管理系统相联系；通过告诉管理层如何支持和授权项目小组，坚持高质量的工作，并且依据数据进行项目的管理，向组织展示如何应用 CMM 的原则和 PSP 的技能去生产高质量的产品。

软件的生产过程及其他的许多子过程、软件的开发者和用户以及系统的使用中存在着巨大的变化和不同，要使一个软件过程对软件生产的改善真正有所帮助，其框架应是由 CMM、TSP 和 PSP 组成的一个完整体系，即从组织、群组和个人三个层次进行良好的软件工程和管理实践的指导和支持。总而言之，单纯实施 CMM，永远不能真正做到能力成熟度的升级，只有将实施 CMM 与实施 PSP 和 TSP 有机地结合起来，才能发挥最大的效力。

11.5 本章小结

一个软件的质量是以用户的需求为基础的，我们说一个软件是符合质量的，主要是指该软件符合用户的需要，符合需求规格说明书。然而，需求具有显性和隐性的特点。显性的需求相对是比较容易控制的，质量往往在隐性的需求上出问题。

软件质量是每个人都想要的东西。因此如何提高软件的质量就成为一个重要的课题，本章从流程、技术、人、组织等多个方面阐述提高质量的方法。Deming 质量原则的应用推动了软件全面质量控制思想的产生，全流程的测试就是在这个基础上发展起来的。在后面章节我们将分析这个概念。

目前业界流行的质量标准有许多，而 CMM 已经成为软件行业的一个事实上的标准。然而要推广 CMM 却是一件难事，因此 Humphrey 的 PSP 和 TSP 理论为推行 CMM 指出了一条道路。

质量改进需要花费成本，因此改进的途径需要视不同公司的规模、业务、财务状况、人员技术水平等多方面综合进行考虑。一般建议中型以上的较大的软件公司实施 CMM 体系。而对于一些小型的软件公司可以采取比较实际的，相对成本较少，且容易操作的方面进行，这些方面大致如下：

- 实施简洁的开发过程体系，根据不同业务特点可以选择瀑布模型、迭代模型等，并在这些模型上进行适当的变化以适应短频快的产品开发特点；
- 提高需求分析和设计方面的技术，例如：原型法技术，分析模式，设计模式，面向对象设计，UML 等；
- 加强文档化工作。文档是经验的保留，对于一个企业要想获得长期的发展，必须加强文档化工作；
- 加强编程规范工作；
- 进行适当的测试工作，建议进行单元测试和系统测试；
- 实施配置管理工作，加强版本控制；
- 开展走读、评审和检视活动，尤其要加强走读，建议进行每日交叉走读活动；
- 进行简单的度量分析活动；建议实施 PSP 活动。

第 12 章 软件验证和确认

Deming 的“PDCA”循环被应用到了测试领域，于是全流程的测试理念产生了。这是一个基于整个开发过程的验证和确认过程。本章，将基于 IEEE 提出的验证和确认标准(IEEE Std 1012—1998)来阐释这个概念。通过本章的学习，应该对以下内容有所了解：

1. 什么是软件验证过程？
2. 什么是软件确认过程？
3. 如何制定 SVVP？
4. 在生命周期过程中，软件验证和确认是如何运作的？
5. 如何根据软件的关键级别选择软件验证和确认的活动？

12.1 基本概念

软件验证和确认(Software Verification and Validation)是在软件的生命周期中对其进行评估的规范化的方法。验证是用于在软件开发生命周期中保证前面进行的活动是满足特定需求的。而确认是在生命周期阶段结束时检查系统是否满足客户的需要。验证和确认工作力争保证软件产品的质量，并使其满足客户的需要。考虑到在产品或开发中实时更改并支持处理，验证和确认使软件管理能够洞悉软件计划和生产的状态。

验证产生于 20 世纪 70 年代，是美国航天局生产高可靠性软件系统实践的一个产物。。在这方面，一个错误可能导致任务的失败和大量时间与财物的损失，甚至会威胁到生命环境。验证的概念包括两个基本原则：首先，软件必须能够正确地执行其被要求的功能；其次，软件不能执行任何不被要求的功能，该功能本身，或者与该功能结合后会降低整个系统的性能。验证的整个目标是保证整个软件生命周期的每个软件产品满足客户规定的需要和目标，这些规定和目标被文档化在软件需求规格说明书中。

验证同时还在软件文档的不同部分和需求规格说明书中相关的部分建立跟踪关系。一个全面的验证努力保证在规格说明书中所有的软件性能和质量需求被充分的测试，并且一旦发生改变时，测试结果可以被重复。验证是一个“持续的改进过程”，并且没有明确的终点。它应该被应用到系统生命周期中以保证配置和操作的一致性。

验证保证软件潜在的功能，并且使软件拥有各种特性：如可移植性等。验证减少了软件可能会隐含的错误(使最终产品在交付前所隐含的错误控制在一个可接受的数字之内)，它提供了一种紧凑的软件开发过程的方法，并且在任何一个时间点上通过提供详细的工程状态来辅助决策者进行管理。当使用验证过程时，要确保软件开发人员正遵循着一种正式的、顺序的和可跟踪的软件开发过程，并执行最少的操作来加强系统的质量。

测试产品的活动更接近于确认活动而不是验证活动。按照传统的说法软件测试过程被

认为是一个确认过程，也就是说软件测试是一个生命周期阶段。应用确认的过程，我们将在每个阶段结束时测试该阶段输出的正确性，包括在需求阶段对需求进行测试，在设计阶段对设计进行测试，在编码阶段对编码进行测试，在编码结束后，系统被验证或测试以确定它的功能及操作的性能

关于验证和确认的关系，Bohem 给出了一个经典的总结^[212]：

- 什么是验证？验证就是要用数据证明我们是否在正确地制造产品。注意这里强调的是过程的正确性。
- 什么是确认？确认就是要用数据证明我们是否制造了正确的产品。注意这里强调的是结果的正确性。

Eaglestone 和 Ridley 把这两个概念进行了集成^[214]，提出：

“我们是否在保持产品的正确性？”

软件验证和确认采用审查、分析和测试等技术来确定一个软件系统及其中间产品是否符合需求。这些需求包括功能特性和质量特性。这些质量特性根据项目的不同而不同。用户对软件产品提出的质量特性是这样标识的：软件拥有足够的性能来满足它的目标，同时又没有引入意外的副作用。

下面罗列了需要确定的质量特性的清单(IEFE Std 1012—1998)。

- 准确性(Accuracy)
- 完整性(Completeness)
- 一致性(Consistency)
- 正确性(Correctness)
- 有效性(Efficiency)
- 可扩展性(Expandability)
- 灵活性(Flexibility)
- 互操作性(Interoperability)
- 可维护性(Maintainability)
- 可管理性(Manageability)
- 移动性(Portability)
- 可读性(Readability)
- 重用性(Reusability)
- 可靠性(Reliability)
- 安全性(Safety)
- 保密性(Security)
- 残存性(Survivability)
- 可测性(Testability)
- 可用性(Usability)

验证和确认工作的目标是要发现缺陷，并确定软件系统是否实现了需要的功能和特性。验证和确认活动主要对软件开发的产品进行操作，并支持以下内容：

- 验证生命周期每个阶段的产品

(1) 遵循前一个生命周期阶段的需求和产品(如，正确性、完整性、连续性、准确性)

(2) 满足阶段的标准、惯例和协定

(3) 建立适当的基础以启动下一个生命周期阶段的活动

- 确认最终的产品遵循了已经建立的软件与系统的需求

典型的验证和确认工作应与软件开发和支持活动并行进行。某些验证和确认任务可能会与开发和支持过程交互到一起。验证和确认工作包括管理任务(如计划、组织和监控验证和确认工作)和技术任务(如分析、评估、审计和测试软件开发过程和产品),其目的是提供工程、质量和软件产品在其生命周期中所处状态的信息。

验证和确认计划在项目的早期开始,目的在于评估验证和确认工作的范围。这一点是非常必要的,它有助于保证在全部工程计划中包含验证和确认工作所需要的资源。最初的软件验证和确认计划同样还可以帮助高层主管比较深入地了解项目的开发和支持计划。它还给高层主管提供足够信息,确定是否采纳计划并监控执行情况。

对于一个项目,弄清楚如何使验证和确认活动与项目的整个生命周期相配合,如何使验证和确认活动与项目实体(如,用户、开发者、购买者、软件结构管理、软件质量保证等)相关联的是相当重要的。有时需要一个完全不同的组织来完成验证和确认工作,该组织通常直接面向软件的购买者。这种情况是指独立的验证和确认。一般来说,在公司内部,验证和确认工作可能由完全分离的组织来完成,也可能由系统工程组织或产品保证组织来完成。验证和确认任务的选取依赖于其项目是如何被组织的(如,不仅要考虑验证和确认工作是如何直接服务于验证和确认目标的,而且要考虑它们是如何服务于整个项目的)。

软件验证和确认任务相互支持并合成一个强有力的工具,它可以完成以下工作:

- 尽可能早地在软件的生命周期中发现错误;
- 确保要求的软件质量被计划并在系统中被实现;
- 预测中间和最后的产品满足用户需求的程度;
- 保证遵循标准要求;
- 确定安全性和保密性功能;
- 帮助避免在产品发布的最后时刻发现问题;
- 提供高可信度的软件可靠性;
- 为管理提供决策的标准;
- 降低操作变化的频率;
- 评估验证和确认活动提出的修改建议的影响。

12.2 软件验证和确认计划

第一个验证和确认工作最早同概念文档评估、需求分析及确认测试计划一起开始。制定验证和确认计划时必须紧密地配合项目的其他部分。计划者应该预见并定期调整软件验证和确认计划(Software Verification and Validation Plan, SVVP),以便反映在整个开发任务中的变化。

计划和计划文档的目的在于有效地利用验证和确认资源,监督和控制验证和确认过

程,确保每个参与者的角色和职责被标识。计划活动产生一个结构良好、彻底和现实的SVVP,它为验证和确认工作的成功提供了一个基础。简要的和详细的目标说明文档保证了所有参与者能够理解将要达到什么目标和不要达到什么目标。

每个项目都要有它自己的SVVP,该计划遵循标准中标识的通用计划(IEEE Std 1012-1986)。具体的计划要能够适应项目的需要。计划可能会因为一些人的调整而实时地被改变。尽管验证和确认工作的基本目标不可能随时间而改变,但还是必须对人的调整进行响应。这些变化可能会影响到开发活动(如在时间表上的重大延迟,不可预见的技术变化)或验证和确认任务(如,资源的流失,不能及时地开发出工具),因此,在计划中应该包括修改计划的过程。

12.2.1 SVVP 步骤

尽管SVVP是整个项目计划不可分割的一部分,但它仍可分解成以下几个步骤:

(1) 明确验证和确认的范围

在开发过程中,应该尽可能早地定义软件任务。这些任务应当考虑一些软件因素(如,关键性程度、完整性、包括开发环境和工具在内的可获得资源等),确定执行什么样的任务需要什么样的决策方法。12.3.1节中给出的关键性分析就是这样一种方法。

在进行下一步工作之前,所有的参与者,特别是用户和消费者,应该对建议的验证和确认任务进行讨论。只有在达成协定后,下一步工作方可进行。

(2) 从总的项目范围中确定特定的目标

详细的、可测量的和可完成的目标是验证和确认工作能否令人满意的先决条件。这些条件是评估和执行验证和确认性能的基础。因为它们需要能够承担得起所制定的目标,因此有必要规定类似负载极限和响应时间这样的可测量的准则。并且在验证和确认工作中应该要有相应的规程和工作用来明确地表达可测量的目标,以便尽可能早地暴露对期望的不明确和有分歧的地方。

识别可完成的目标是很重要的,不切实际的目标降低了计划的可行性,以及整个验证和确认工作本身的可信性。一个现实的计划增强了项目参与者的信心,也确定了验证和确认工作的价值。

(3) 在选择验证确认工具、技术以及准备计划之前分析项目的输入

在项目各阶段中对可以获得的具体产品、信息和文档进行输入分析。图12-1中给出了类似的这些的产品输入实例,输入可分为两种类型——项目刚开始时的产品和信息以及项目下个开发阶段中将要得到的产品和信息。计划阶段的信息有的是与产品相关的,有的是与项目相关的。与项目有关的信息包括预算、人力和其他资源、开发时间和里程碑,以及其他需要考虑的与项目有关的细节性问题,例如契约性需求等。与产品有关的信息包括概念文档、需求规格或其他用于产品开发的正式规格。需要注意的是这些产品的术语是随着用户和开发环境的不同而变化的。

如果可以获得相同或相似软件以往版本的缺陷数据,这将会是非常有帮助的。可以通过类似缺陷度量和分析异常情况等工作来获得这些历史数据。在计划验证和确认工作时,需要预测和分析软件特性以及将要出现的缺陷数量,这些工作对验证和确认任务的选择和

确定完成这些任务的资源是很有帮助的。

(4) 选择工具和技术

挑选可用于项目的工具和技术(如那些支持可跟踪分析的工具和技术),并收集足够的信息来为决策提供选择。对于技术而言,需要的信息包括需要的输入、应用的范围、资源和需要的技巧等。对于工具而言,需要的信息包括工具和相关文档的可获得性、可应用性以及操作和培训所涉及的资源需求等。

在选择验证和确认所采用的工具和技术时,应该考虑有关软件开发环境、方法和工具等具体信息。技术和工具需要时刻兼顾开发产品的模式、结构和进度。此外,尽管在一些情况下,使用单独开发和特制的工具可能是合适的,但一般来说,使用具有相似功能的通用工具更经济有效。

为了满足项目的特定的约束和需求,需要得到一套与之相适应的工具和技术。在这个过程中,需要考虑个人的技能和培训,以及其他必要资源的可获得性。

(5) 开发计划

分析上一步的结果并准备一套详细的任务以便满足验证和确认的目的、目标和约束。

12.2.2 SVVP 的 7 个主题

为了对每项任务进行清楚和明确的描述,IEEE 标准对 SVVP 定义了 7 个主题。没有必要对每个主题进行单独说明,这些主题的结果和方法由计划者来阐明。

1. 验证和确认任务

该主题要求定义和描述每个阶段的任务,以及任务对完成验证和确认目的有什么贡献。SVVP 必须为验证和确认的管理以及每个生命周期阶段提供一个最小的验证和确认任务集。这个任务集中的任何一个或所有任务都可以在非关键软件中使用。有关软件验证和确认的任务分析将在 12.3 节描述。

当计划验证和确认任务时,应当考虑以下问题。

- 确定每个任务的范围。决定关注整个系统还是一些子系统。
- 确定每个任务的验证和确认目标。决定是用验证(Verification)、确认(Validation),还是结合两者。选择的每项任务必须满足项目级别验证和确认目标以及质量保证目标。定制的验证和确认任务有以下目标:
 - ✓ 验证生命周期内各阶段的软件的规格和每项活动的输出是完整的、正确的、清楚的、一致的,能准确满足前一个阶段的规格要求。
 - ✓ 评估产品生命周期内每个活动的技术的充分性(如:评估替代方案),并监督重复活动直到找到合适的解决方案。
 - ✓ 跟踪生命周期的每个阶段开发的规格,与前一个阶段的规格进行对比。
 - ✓ 准备和执行测试,验证产品是否满足所有的应用需求。
- 确定验证确认活动和其他活动之间的依赖关系(如,此项任务从哪项任务取得输入,为哪些活动提供输出)。
- 确定软件的关键性。决定软件的关键性级别。IEEE 标准为关键软件定义了最小

的任务集(参考12.3.1节获取更详细信息)

- 确定项目环境。评审和评估项目计划,包括时间表、变更控制、配置管理和资源。在挑选哪些需要开发新工具和方法的任务时,这些资源可能会是一个约束因素。
- 确定产品和开发活动的复杂性。估计和评估产品的复杂性、产品的风险和开发的复杂性(如项目规模大小和组织等)。
- 确定预计的缺陷数量和类型。可能的话,估计一下缺陷的数量和类型。这些信息往往可以从经验或理论上得到,可以用来评估验证和确认活动的可行性和带来的好处
- 确定有经验的员工和专家。顺利执行验证和确认任务需要的时间和培训是需要考虑的一个重要因素。

2. 方法和标准

该主题用于定义执行验证和确认任务所需要的方法和标准,描述执行每项任务的具体方法和过程,定义评估任务结果的详细标准。一个验证和确认任务可能会用到不止一种方法,例如源代码评估可以进行代码检视和算法分析;一个具体的方法可能会应用到多个任务,例如时间分析可以在系统设计和代码评估时都用到。

可以用图、表或其他表达手段来表示验证和确认任务之间的关系,在选择这些手段时,需要考虑以下因素。

- 系统的关键特征;
- 达到任务目标的综合方法;
- 组织和个体采用的方法;
- 项目预算、时间表和资源;
- 任务的输入,如:
 - ✓ 开发任务输入的标准。
 - ✓ 输入的抽象级别。
 - ✓ 输入的格式。
 - ✓ 输入的期望内容。
- 任务的输出、每项任务的目的和目标、形式和格式以及期望内容
- 用此方法期望可以发现的缺陷(如标准、算法、新技术或不同特征的误用)

描述每个方法和其过程。陈述该方法是如何满足SVVP要求的验证和确认任务目标的。陈述方法选用原因的所有信息(例如该方法通常能够发现这种工程问题中的一个特别的错误)。如果该方法仅适用于验证和确认活动的某些输入部分,那就把这部分定义清楚。如果一个任务中要用到多个方法,为保证整个任务完成,需要把方法之间的关系和遵循前后过程顺序描述清楚。

定义评估任务输出的标准。包括任务执行完成的判断标准(例如结构化测试只有在路径覆盖达到75%,判定覆盖达到100%才能结束)。

3. 输入和输出

该主题用于定义每项验证和确认任务的输入需求,指明每项输入的来源和格式,以及定义每项验证和确认活动的输出并指明每项输出的目标和格式。验证和确认任务的输入有两类:需要验证和确认的产品,验证和确认的计划、过程和以前的结果。开发的产品是验证和确认的对象。验证确认计划、过程和结果指导验证和确认活动。验证确认计划者需要考虑这两类输入。当执行验证确认活动时,需要给定的输入形式和格式。在许多情况下,对格式(例如 IEEE 标准、美国军方标准)和形式(例如存储媒体、特别的字处理器),交付的输入等有约定的合同需求。

当需要指定任务的输入时,要考虑以下细节:

- 确定输入的来源。也就是具体的组织或个体;
- 确定交付输入的时间表。如果验证确认计划没有综合考虑其他的计划,输入的交付时间就可能不确定。如果需要,时间表应有充分的时间进行重新制定和分配;
- 在任务或方法有多个输入时,需要考虑输入之间时序或逻辑的关系。例如,一项任务开始时,可以没有产品 A,但必须有产品 B 和产品 C;
- 确定输入的状态(例如原始或正式版本)。为尽早检测和修正错误,推荐对原始资料进行评估。有时候,这样做不到(例如合同或政治原因),同时经常无法预计(如资料变化太快);
- 确定输入的形式(例如文件使用 5.25 英寸软盘)和格式(例如 ASCII 文本文件)。在其他任何软件规格中,这些是不能含糊的(例如软盘可能是 3.5 英寸、5.25 英寸或其他,记录和字处理格式也可能有许多)。

在定义每项任务的输出方面,(至少)有 3 种人需要用到**验证和确认任务的输出**:执行其他验证和确认任务的(包括其他生命周期阶段的)人员,**执行开发任务的人员**和管理者。这 3 类人都需要就产品状态反馈技术方面的内容,决定下一步**开发如何进行**。他们对具体信息的需求是不相同的。

- 执行其他验证确认任务的人员需要详细的技术信息;
- 开发人员希望反馈有关产品验证和确认的信息;
- 管理者需要有关时间表、资源使用、产品状态和风险等信息。

验证和确认的输出包括任务报告、不定期报告、测试文档、计划和阶段总结报告等。验证和确认输出是验证和确认任务输出的具体的产品,需要仔细进行计划。当需要指定验证和确认任务的输出时,应当考虑以下内容:

- 每项任务输出的形式和格式(与输入一样,输出应该有统一的标准);
- 每项任务输出的目标(具体的个体或组织单位);
- 每项任务输出的交付时间表(输出应按时交付);
- 每项任务输出的状态(原始或正式版本);
- 每项任务输出的生命周期(是否为下个阶段的输入、还是归档的参考);
- 配置管理对每项任务输出的要求。

4. 进度表

该主题制定验证确认任务的进度表。确定每个任务开始和结束之间的具体里程碑，每个里程碑的具体输入和输出物。制定进度表是为了在时间和事件进程的约束下，通过建立和组织资源来满足确定的目标。为了确保成果，计划者必须认识到验证和确认任务的进度时间表是整个项目大进度时间表的一部分。在进度表实施之前，必须确定能够得到的资源以及项目其他接口部分带来的时间和事件进程的约束。当外部环境建立以后，就要确定验证和确认任务的具体需求。所有与进度时间表冲突的因素都必须解决掉。在建立初始的进度时间表时，时间进度表的重复属性是一个要考虑的因素，在整个项目的生命周期也是一样的。大部分涉及软件的项目在产品发布前都会经过一些实质性的改变。尽管在项目初期极少了解这种改变的程度，但是，进度表必须有响应这种改变的机制。

在软件生命周期的所有阶段，都涉及到验证和确认任务。按照组织的职责分工，验证确认计划者必须明确生命周期各阶段，需要哪些组织，进度表和文档接口等。必须牢记所有的验证和确认任务都必须与项目的活动同步（如需求分析不可能在需求文档完成后才开始）。

当针对每个生命周期阶段，根据部门职责制订进度时间表时，需要考虑如下内容：

- 所需输入的预定日期，如规格、详细设计、源码或其他数据。
 - ✓ 制定输入的完善标准（如进入配置控制的需求）。
 - ✓ 制定转换所需要的进度时间表（如把需求写成点子格式，不同格式之间转换等）。
- 确定在生命周期特定阶段，参与内部评审的进度时间，保证在产品的开发过程中，验证和确认所关心的能够得到强调，如可测试性等。
- 针对项目管理，程序要求确定每项任务输出的预定日期时间。
- 接触接口部门的关键人物，征求意见。

当与其他部门的需求明确以后，生命周期各阶段的具体验证和确认任务就必须确定下来。在确定任务所需资源前，要确定各挑选出的验证确认任务的具体完成标准。根据以前的项目经验、各种产品和项目的自动工具或者外部的咨询，估计所需资源的满足情况。这些估计可以通过工具来改进效果。尽管这些估计在项目初期存在一定程度的不确定性，在项目实施过程中，当信息越来越明确时，应该及时更新。

在制定生命周期阶段的验证和确认任务进度时间表时，需要考虑以下内容：

- 制定验证确认的输出和验证确认报告的进度时间，为开发者提供及时和有效的反馈；
- 验证和确认管理者要考虑对输出的评估和内部评审；
- 定义明确的验证和确认任务停止标准；
- 定义变更后回归测试工作量的标准。

通常，项目进度表有许多不同格式（如甘特图、PERT图、CPM）和分析流程。采用的方法必须与其他项目一致。鼓励使用自助工具。必须明确实施进度时间表改动的标准和流程。

5. 资源

该主题用于定义实施验证和确认任务的资源,明确资源的类别(如人力投入、设备、仪器、进度、旅行、培训等)。如果验证和确认任务用到工具,指明工具来源、可能性和使用需求(如培训)。通过任务承担者姓名或其他名词(如工作名称或分类)来指定人力资源。然后,提供充分详细资料,说明选择的人员有能力保证完成任务。

当计划验证和确认的人力资源时,需要考虑:

- 人员数量或等价时间(如两人一半时间)。人员能够及时到位是人力使用的理想状况。人力供给不足是选择任务、方法、制定进度时间和任务覆盖度的关键约束;
- 需要的技能等级和专门的经验。清晰定义特殊需求(如对安全操作系统实施需求跟踪的高级分析员)。当预计验证和确认任务的效率和评估历史数据相关性时,考虑人员的经验和专长。一般没有经验的员工会比有专长的员工引进更多的缺陷,因此必须描述出执行任务所需要的最小级别。这就给任务分配提供了最大的灵活性。如果所需经验的人得不到,就要安排员工进行培训。标明需要人力的高峰时刻。

指明执行任务时所需要的手动或自动的工具,例如 Checklist、文档、标准化的验证和确认过程等。当考虑使用手工工具时,需要考虑以下内容:

- 工具必须适合任务、充分可用并能提供有意义的、具体的结果,否则就要考虑采用其他工具。比如,一般的 checklist 必须根据项目情况来剪裁;
- 对给定的任务使用工具前必须被评估,以保证有效。

自动工具包括软件工具、模拟器(软件、硬件或软硬混合物)、测试环境、测试桩和电子通信工具的使用。与手动工具一样,提供工具详细的使用方法是做好计划并从中获益的重要条件。当计划使用自动工具时,需要考虑以下内容:

- 指明特定的工具(包括版本标识符)和工具使用时需要的准备工作(如开发、剪裁、适应性等);
- 指明工具使用需要的资源,包括计算机、其他硬件、数据库、测试桩、仪器和相关的培训及维护;
- 为使用工具的许可证或合同做好计划;
- 考虑好如果无法得到工具,如何达到任务的目标。

资金资源也要被提到,尽管通常在基于任务的 SVVP 中看不到。应包括特别的资金考虑,如执行某一具体任务时所做的平衡。

进度表还要包括任务的旅行需求(如参加设计评审会议)和交叉旅行计划。其他需要提到的资源有特殊过程,如安全控制。通常,这些内容仅在某些特殊任务的过程中与整个项目其他部分不相同时才在该任务范围内提到。

6. 风险和假设

该主题识别与验证和确认任务相关的风险和假设,包括进度时间表、资源、方法,为每项风险制定一个风险计划。

任何计划都是基于一些假设。这些假设构成了计划中的风险因素，如果假设是不成立的，项目自然无法按计划进行。许多情况下，如果这些假设被记录在案，并制定了风险计划，往往可以降低这些风险。风险计划必须是实事求是的。除非受风险级别和验证确认任务关键级别的要求，一般不要求详细描述风险计划的细节。当在记录风险和假设时，应当考虑：

- 软件项目进度时间表。基本的假设是：软件或软件组件和交付物将按进度时间完成、控制、更新和提交。通过记录哪些任务在输入或资源上依赖于其他任务，交付延迟而导致项目混乱的潜在可能性将大大减少；
- 软件项目的复杂性。当软件结构、组件或交付物技术非常复杂时，就存在项目早期没有充分评估该软件的风险，将会导致在后期错误不断增加的风险；当软件项目的组织结构复杂时，就存在沟通不够、导致接口和文档错误的风险；当验证和确认任务比较复杂且与现有人员和其他资源相关时，就存在进度无法保证的风险；
- 资源满足的充分性。当人力资源不足（如技能、经验等）或资料不充分（如计算资源、测试桩、负载生成器、代码表态分析器等），就存在验证和确认任务执行不充分的风险；
- 有无标准的产品开发过程（如配置管理、评审、标准等），这些会影响预期的缺陷数，同样会影响验证和确认的人力投入预算。一个基本假设是，需要的话将会有及时的非正式的说明和异常报告。

其他软件在验证和确认过程中可能出现的风险有：

- 开发并第一次使用软件工具；
- 使用 COTS (Commercial-off-the-shelf)；
- 对大规模和复杂软件多次使用 V&V 技术；
- 生命周期模型的改变。

7. 角色和职责

该主题制定组织单位或个人执行验证和确认任务的职责。针对具体任务，给某个或多个组织单位分配具体职责，包括独立制定计划、执行、报告、评估、合作和管理。许多情况下，一些角色将被定义为任务的输入或输出。比如代码走读，有人提供代码，有人分配代码给评审者，另外一些人组织评审队伍。一个人或组织单位可以指定担当多个角色。

根据承担任务者的姓名来组织将使一个主进度时间的结构和协调变得简单。比如，一个人任务超负荷或使用错误时，可能很容易被发现。通过阅读 SVVP，可以很容易地指定参与项目的个人及其角色。另一方面，根据姓名来安排也有缺陷。如果一个人换了，与其相关的任务可能被丢失。对大的或长期的项目，有许多人参与，在一个相当大的计划中，有太多的不确定性。因此，通过组织和工作分类来组织员工是个好办法。可以用矩阵、表格、图等来给部门和工作类型或个人分配验证和确认任务。计划必须尽可能具体（如指定完成某任务的具体人数）。合适的时候，作为一个例子，可以包含个人的履历（可能在附录中）。对大的项目，职责的分工可以放在文档的附录部分。

12.3 验证和确认任务分析

软件验证和确认整个过程包含了众多活动，这些活动彼此关联，共同组成了一个跨越各生命周期阶段的连续活动。这些活动使得验证工作得以更有效地执行。SVVP 制定者应该考虑在生命周期的整个过程中集成和分配资源，以便连续活动中的所有任务的计划和执行能够有效地开展。表 12-1 给出了这些活动之间的关系。

表 12-1 验证和确认任务关系表

任务作用	任 务			
	可跟踪性分析	评 估	接口分析	测 试
关键性分析	标识要跟踪的关键特性	标识需要评估的关键区域	标识关键接口	建立测试的优先级
可跟踪性分析		确定需要被评估特性的具体位置	标识依赖构件之间的接口以及通过接口传递的信息	跟踪测试文档，通过确定怎样分配功能来制定测试策略
评估	保证功能分配的结构有效性		建立适当的并能正确实施的接口	标识需要动态确认的属性以及测试所要进行的程度
接口分析	建立可跟踪性的需要并保证依赖成员之间的通信	标识成员之间的关系，及每个成员的输入和输出要求		建立集成测试计划的框架
测试	确定可跟踪性和功能分配	确认评估	确认接口是正常工作的	

12.3.1 关键性分析

关键性分析是一种方法，研究如何把验证和确认的资源分配到软件的各个基本部分。关键性分析的目的在于保证资源的有效利用，尤其在需求资源得不到满足时，有必要在整体上分析整个系统的资源分配策略。简单地说，关键性分析，就是一种把验证和确认的资源分配到软件中最重要部分的系统分析方法。

关键性分析与关键软件概念相关。按照标准定义，关键软件是指那些其失效会影响安全问题或者会导致经济或社会损失的软件。如果一个系统满足关键性的定义，标准要求至少有一组最少任务必须被执行。但是要执行到何种程度，在标准中并没有提到。对于关键软件，关键性分析提供了如何做出这些决定的更加系统的方法。

关键性分析由 4 步组成：

(1) 为开发的系统建立关键性级别。关键性级别可以由高、低，或者高、中、低，或者由 1 到 N 的一组数字构成。大多数情况下，有二至三个级别就够了。

对于一个给定的系统，经常有几种不同类型的键性。比如安全性、系统任务、金融

风险、技术复杂性等，是常见的关键性类型。

定义每种级别和关键性类型。如，安全性级别“高”可以指如果系统失效的话会导致人员伤害、死亡、财产损失等；级别“中”指系统失效会导致设备的损坏；级别“低”指不影响人员和财产。为每个关键性类型定义类似的级别。级别应该定义得清楚明白、无歧义，这样不同的分析员各自独立工作时可以得到相同的判断。但达到这个目标比较困难。

在IEEE标准中，使用软件集成级别方法来对软件关键程度进行界定。为了计划验证和确认过程，软件集成级别在开发早期就被分配，最好是在系统需求分析和体系结构设计时进行。软件集成级别可以分配到软件需求、功能、功能组或软件构件、子系统中。已经分配的软件集成级别也有可能随着软件的演进发生变化。所采用的设计、编码、程序和技术将影响软件的关键程度以及相关软件的集成级别。在软件开发过程中，软件集成级别分配将在验证和确认的关键程度分析活动中被持续地修正和评审。

IEEE标准把软件集成级别定义了4个级别，具体如表12-2所示。

表 12-2 软件集成级别定义

关键程度	描述	级别
高	所选择的功能将影响系统关键性能	4
重要	所选择的功能将影响系统重要性能	3
中	所选择的功能将影响系统性能，但工作区策略可以对其进行补偿	2
低	所选择的功能将影响系统功能，但只是给用户操作带来不便	1

(2) 定义满足系统特征的需求。这一步必须是开发过程的一部分。有时候，完整的或实用的需求并不存在，因此需要进行开发。完整和可信的关键性高的级别由哪些组成，在这点上，开发者、用户、验证和确认代表以及所有程序管理者必须达成一致。细节的实际级别依赖于系统的复杂情况。给每个定义好的需求分配一个数字，并且与其在需求规格文档中的原始需求的段落编号相对应，形成交叉参考

(3) 针对每种关键性类型，为每个需求定义关键性级别，把多个级别合成简单的关键性级别。如果系统的需求不清楚或者没有量化，分析者分配关键性级别就比较困难。这可能重新提炼或校正需求。差别不大的情况下，保守的作法（尤其在设计关键软件时）是给高一点的级别。

关键性的类型比较多时，不同类型的级别可以不同。例如，假设系统有几种关键性，技术复杂性、安全性、金融风险等。技术复杂性的级别可以是低，安全性的级别是中，金融风险的级别是高。用下面两种方法中的一种，可以把这些不同的级别合成一种通用的级别。

第一种方法保守一些，推荐在系统关键性设计时使用。在这种情况下，不同类型的關鍵性级别中，需求的关键性级别是最高的。第二种方法是把不同类型的關鍵性级别进行平均。加权的平均方法也可使用。舍入法是解决部分平均问题的保守方法。在上面的例子中，直接平均的级别是中。如果比起其他两种关键性类型的一种，技术复杂度的重要性增加一倍，用舍入法也会得到级别中的结果。

(4) 针对每个需求，使用关键性级别来决定验证确认方法适用于需求的强度级别。根据需求的关键性，可能需要更加深入的分析。例如关键性级别比较高的需求，可以采用全

面的技术和专用的开发工具。关键性较低的只需使用一般的分析，或者在资源严重受限的情况下什么都不做。

SVVP 是一个主观的过程。在此过程中，关键性分析方法用来指导计划者对大多数的关键性需求进行全面的分析。这样可以减少重要需求被忽略或分析程度不足的机会。由于过程的渐进性，计划过程中的“决定”和“平衡”就更加可视化了。因为每个计划的假设和决定都可以定义和分析，这种可视性就能保证对计划的评审和评论以建设性的方式进行。

12.3.2 可跟踪性分析

IEEE 的验证和确认标准要求需求、设计和实施阶段进行可跟踪性分析。可跟踪性就是标识原始需求和它们的结果系统特性之间关系的能力。它容许一个系统的生命周期中在由需求的分解和提炼所产生的惯性关系网络中向前和向后跟踪。可跟踪性提供了连接元素的线索。当由一个元素跟踪到另一个元素，从那个元素又跟踪到另一个元素，这样一个因果链就形成了。

可跟踪性使特性验证在概念和需求规格完成下面活动时就开始了：

- 概念和需求规格推进到设计规格；
- 概念和需求规格在代码中实现；
- 概念和需求规格在测试计划和用例中包含；
- 概念和需求规格在最后系统中提供给用户和消费者。

可跟踪性，借助于向前和向后跟踪，可以方便地构成高效的测试计划，并允许对覆盖了功能和设计需求/特性变化的测试用例结果进行验证。

当从所有软件需求跟踪到概念文档，再对系统需求进行验证时，就是跟踪性分析，这包括从软件需求到整个开发的输出、用户文档、测试文档等。每个跟踪按一致性、完整性和正确性来分析，以便验证全部的软件需求是否在软件中得以实现以及是否与正确的设计、代码和测试信息相联系。

在可跟踪性分析中，有些错误是显而易见的，如没有设计元素的需求、没有代码的设计元素，或者是这些情况反过来。分析者检查每个跟踪路径以确保被连接的部分是正常的。为了完成这些，分析者必须理解需求和设计的意图。

可跟踪性分析可用来支持配置管理、测试覆盖率分析、验证和确认结果分析、回归测试、关键性评估，以及验证和确认的管理决策。此外对于好的软件工程实践来说，可跟踪性分析在评价软件开发的工作中也是有用的。

进行可跟踪性分析的目的是保证：

- 规格说明书中的每项需要都被正确标识；
- 从前一阶段到当前阶段的全部跟踪线都是连续的；
- 当前阶段与前后阶段之间跟踪是一致的；
- 从每个规格开始的前向跟踪完全支持规格；
- 每个当前的规格或特性都完全被可跟踪的前任规格所支持。

12.3.3 评估

评估调查每一项的价值，协助帮助保证系统满足它的规格。在生命周期的整个阶段都需要许多人来完成评估，既对中间产品也对最终产品进行评估，对一个系统既可能进行全面的也可能进行选择性的评估。

评估可以发现在不同产品及其相关部分中的问题。这些问题与系统满足其预先设定的、用户需要使用的功能有关。为了使产品具有使用价值，评估工作可以保证下面几点：

- 产品符合规格；
- 产品正确；
- 产品完整、清楚、一致；
- 考虑了合适的替代品；
- 产品遵循所用的适用标准；
- 产品满足所有规定的质量特征。

评估不仅是找错，而且它还通过推荐以下方案来帮助确定软件开发的过程。

- 阶段的连续性；
- 先改正规格再继续；
- 回溯几个步骤并改正错误；
- 在系统的某部分优化之前，进行附加的评估，如进行模拟；
- 监督一个质量可疑条目的执行过程；
- 更改方法或工具；
- 更改时间表；
- 进行机构决策，如附加资源、附加培训或变更任命。

评估用于全阶段和软件产品的全部类型，包括用户文档、手册和其他项目文档。这些文档具有很多形式，如文本或图形表示，而且采用多种媒体形式，如纸、磁带、磁盘、计算机文件等。如此多的产品类型和形式需要大量不同类型的技术来完成和管理软件评估工作。

在评估系统产品时，需要随着开发阶段的前进而不断改变重点。因为一个产品从概念到产品是不断前进的，所以它的功能需求要首先确定，然后建立设计元素的组织与结构，这样可执行的工序就都准备好了。在开发初期，像“是什么和为什么”、“如果怎么样”、“更好或最好”这样的问题是主要的；在开发后期，“怎样”、“正确性”和“一致性”之类的问题会更多一些。

在选择评估策略的过程中，SVVP 制定者首先利用系统的性质和特征来选择评估技术。计划者需要质询系统的功能，例如，如何阐述它们，可能遇到什么样的问题。其次 SVVP 制定者根据系统特有的品质特性来选择辅助的评估技术。第三，SVVP 制定者需要基于特殊的验证确认关系和经验来选择其他评估方法。

考虑下面 3 种方法可能对选择评估技术会有帮助。

- 静态分析规格(如评论、检视、结构分析)；
- 动态分析预期的软件行为(如模拟、建模、屏幕仿真、分支执行分析)；

- 正规分析(如数学校对)。

所有这些评估类型都可以相互利用,以便提供一个强有力的工具。一个特定的评估类型的选择依赖于所需要的结果、可用的工具和价值交换。

静态分析是通过对软件产品进行手工和自动的检查来发现错误和不足,而不用执行软件代码。例如,静态分析可以用于分析规格的形式与内容,这通常是很直接的。它不断地利用不同的评价类型,将其应用到规格的全部标准,它发现可以预先排除的错误,或者替代其他的分析形式,或者避免对产品的不完全测试。静态测试集中在形式、结构、产品、内容、接口、从属物和使用环境上。静态分析可以用于开发的所有阶段,以及所有提交的文档。

动态分析需要执行软件本身或其模型、简化版本,以此来确定它的一些属性是否有效。

12.3.4 接口分析

当消息通过一个接口时(例如硬件和软件、软件和软件、软件 and 用户之间),消息的一部分可能会丢失,消息的内容可能会被改变。接口分析的任务就是确定接口的完整性、准确性和一致性。在设计和开发阶段,接口需求分析应在功能、物理和数据接口层次上定义和分析。接口分析的目的在于评估软件提交物(包括需求、设计、编码)正确、一致、完整和准确说明了接口需求。

接口分析是和数据流联系在一起的。信息源在保证该数据流是否可用上是重要的。关于接口数据接受方必须保证每一个接口是必要且充分的。接口的正确性依赖于接口的用途,形式和内容。

接口分析必须关注以下 3 个接口领域。

- 用户接口:它涉及分析人和软件产品的接口形式,例如,所需的屏幕格式、保护机制、页面布置、报告内容、输入和输出的时间间隔。
- 硬件接口:它涉及分析系统软件和硬件之间每一个接口的逻辑特征,确定电子器件、固件、通信装置和输出装置。再确定接口的适用的标准,检验当前的应用接口。
- 软件接口:它涉及分析本软件和系统软件(数据库管理系统、操作系统、库程序包),本软件和应用系统之间的接口。确定系统软件的可应用标准和与应用系统之间的接口。检验它们之间的软件接口是否正确。

通常接口分析考虑以下几点:

- 接口是否在技术上有足够的和深刻的理解。
- 是否所有的数据要素被清晰定义。
- 是否所有的限制和约束被清晰定义。
- 是否所有不同的接口都被说明和正确描述。
- 是否所有的软硬件接口被量化确定,例如,消息单元、速率 bps、消息格式、优先级规则、字长度、时限和协议。
- 是否所有软件接口功能和所有的数据接口都被量化确定,例如,数据时限、数据

定义、数据格式、优先级规则、消息内容和通信协议。

- 接口的性能需求有没有定义，限定有没有明确。
- 接口的关键程度是否被充分考虑。
- 如果接口降级将带来什么影响。
- 接口是否可测和可维护。
- 是否关于接口的所有的适用的标准都被确定，包括当前的应用软件到其应用环境。

12.3.5 测试

测试是分析软件的过程，通过测试检测出现有软件和需求条件之间的差异，评估软件的特性（注：在验证和确认标准中提到的测试是指传统意义上的软件测试）。

在软件检查和验证的过程中，测试可以被定义为支持验证和确认目标的测试。这种测试的目标不同于开发人员的测试。例如：验证和确认测试可以集中在一个窄的、关键的区域或保证开发者计划测试的一致性。

SVVP 中定义的有关测试的范围和组织职责依赖于特定项目。验证和确认目标确定，被执行的测试将支持这些目标。要考虑除这些软件检查和验证外，可能还有别的测试活动，也可能涉及到别的组织。作为验证和确认的一部分，测试可以由软件开发者或非软件开发人员来做。

12.4 生命周期各阶段活动

尽管许多任务跨越了生命周期阶段，但是大部分的 SVVP 还是按照特定的生命周期阶段来制定和管理任务。另外，IEEE 标准把管理作为一个阶段来定义。当然，管理并不是一个单独的生命周期阶段，而是跨越每一个生命周期阶段的一种活动，并且与所有活动紧密联系，形成一个统一的、有意义的程序。软件验证和确认的阶段活动情况可以参考图 12-1。

12.4.1 管理阶段的验证和确认

验证和确认的管理贯彻于所有的生命周期阶段。软件开发可能是一个循环或重复的过程。针对循环或重复开发过程引起的软件修改，验证和确认工作必须重复旧的或进行新的验证和确认任务。在验证和确认的输入和输出中如果发现了错误，就要重复验证和确认任务。

验证和确认管理的主要任务是计划、评审和监控。在 IEEE 标准中，把这些职责分解成 4 个验证和确认管理的基本任务：SVVP 制定、基线变更评估、管理评审和评审支持。从理论上来说，管理的职责是确保验证确认活动和其他软件开发活动之间能够正向地、成功地配合，从而保证开发出无缺陷的软件。在制定管理任务时还需要认识到验证和确认任

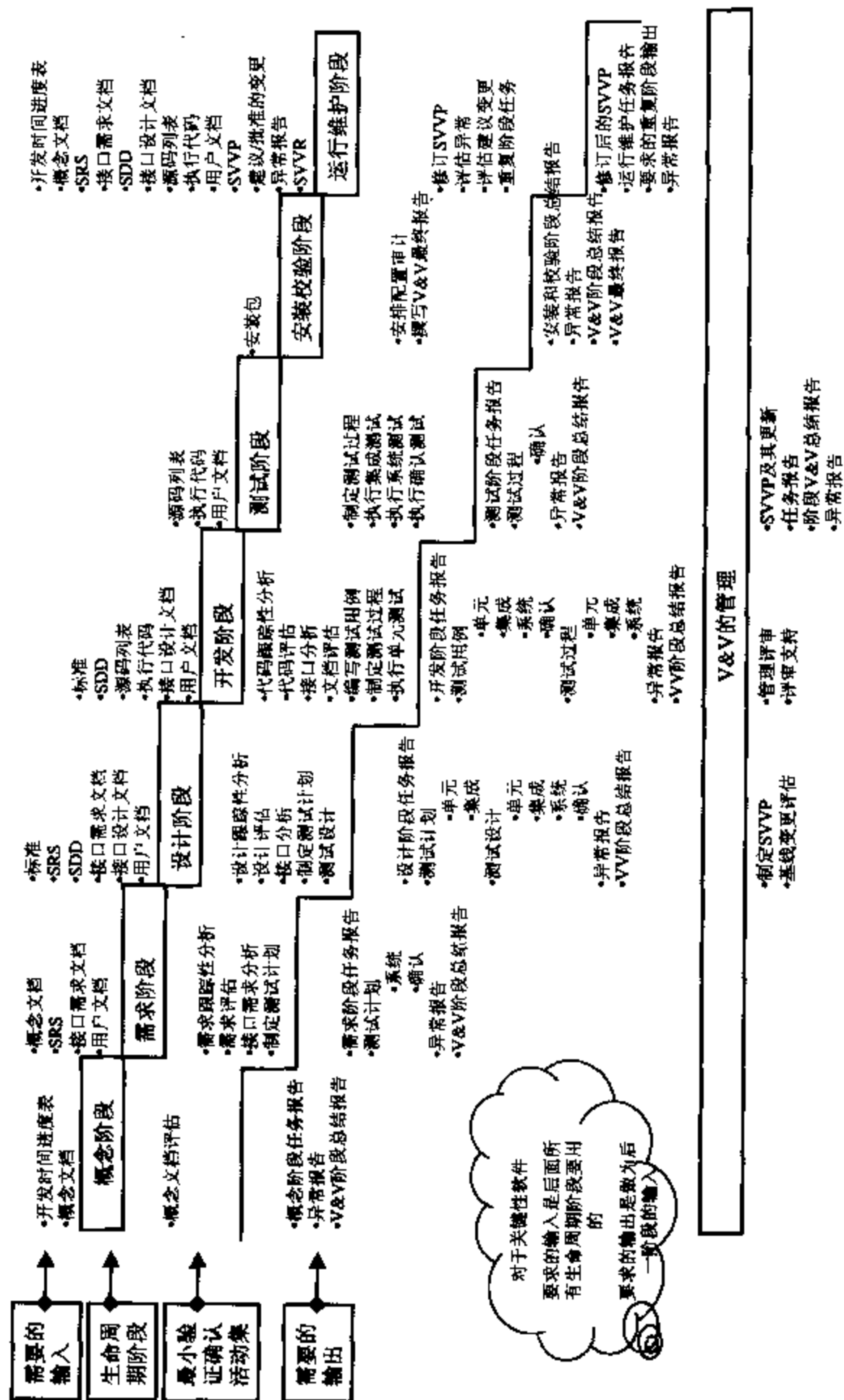


图 12-1 软件验证和确认活动总图

务的自然迭代特性(例如:决定什么时候对修改了的软件产品进行重新分析、什么时候修改 V&V 计划以便反映开发过程中出现的变更)。

对所有的软件,验证和确认的管理应该包括如下的最小任务集:

- 生成软件确认和验证计划;
- 基线变更评估;
- V&V 管理评审;
- 管理和技术评审支持;
- 与组织和支持过程的接口。

12.4.2 概念阶段的验证和确认

概念阶段将明确系统创建的原因。这些反映系统特征的概念将被细化,并列出它的所有目标和在技术与商业约束上的风险。在概念验证和确认活动中,系统体系结构被选择,系统需求被分解到硬件、软件 and 用户界面等子系统。概念验证和确认活动将主要关注系统体系结构设计和系统需求分析。验证和确认的目标是验证系统需求的分配,确认所选择的解决方案,确保不成立的假设不被引入方案之中。

概念阶段的评估应该明确系统满足用户需求的系统目标,系统在技术上、商业上所期望达到的优势。这些目标可以用多种方法进行表述,而且可能包括一些关于需求、商业案例、可行性研究和系统定义方面的语句。

验证和确认创建了有关风险和约束的条目,列出了所有可能妨碍系统开发的有关技术、商业或政策上值得考虑的因素。其中还应该包括初始的计划,并且应该列出对每一次迭代所期望的人员配备、时间和花费。另外,管理系统和它的开发过程的任何规则和政策都应该预先规定好。同时给出替代的方案,并分析其利弊。

该阶段包括的任务有:

- 概念文档评估;
- 危险程度分析;
- 硬件需求/软件需求/用户需求分解分析;
- 可跟踪情况分析。

12.4.3 需求阶段的验证和确认

需求阶段是软件生命周期的一个组成部分,在这个阶段,所有的需求,如软件产品的功能性和非功能性的能力要求,将被定义和文档化。需求阶段的主要输出是软件规格说明书(Software Requirement Specification SRS)。在 SRS 中需要准确地描述软件的任务,或者说是该软件准备做什么。SRS 应可以向后跟踪,回溯到用户需求和概念文档中所定义的系统概念。它应可以向前跟踪,从成功的开发阶段和实现跟踪到设计、编码和测试文档。它还应该同软、硬件的可操作环境相适应。在需求描述中应该为后面的设计和实现提供质量和数量两个方面的约束。有 5 类需求:功能类、外部接口类、性能类、设计约束类和质量属性类。需求的验证和确认活动主要关注软件需求分析。其目标是确保正确性、完整性、

准确性、可测试性和需求的一致性。

该阶段包括的任务有：

- 可跟踪性分析；
- 软件需求评估；
- 接口分析；
- 危险程度分析；
- 系统测试计划 V&V；
- 确认测试计划 V&V；
- 配置管理评估；
- 冒险分析；
- 风险分析。

12.4.4 设计阶段的验证和确认

设计阶段是软件生命周期中的一个阶段，在这个阶段中进行框架、软件单元、接口和数据的设计，同时这些设计被记录在文档中，并且还需要验证它们是否满足需求。在此阶段中，修改设计中的错误会大大减少后续编码阶段错误的发生率，降低软件生命周期中产品和项目的风险。设计的验证和确认也间接地提供了一个发现以前需求中未检查出的错误的机会。

虽然简单产品的设计只需一步进行，但一般说来设计应该分为多个步骤。第一层的设计指定了框架特性(例如子系统及其接口)。后续设计通过增加细节进行扩充，直到子系统可以开始编码。设计可以用很多形式表达，包括文本、图形描述、编码描述、伪代码表达或组合起来的方式及其他。

除了解决如何组织和建立所需系统的问题外，设计者还应有一个范围广泛的质量目标，包括：

- 在所有的设计层次跟踪需求，保证需求不多不少；
- 进行结构化设计，使其符合系统目标和产品质量特性；
- 描述所有的硬件、操作和软件接口；
- 对设计所遵守的可用标准、惯例、约定编制文档；
- 设计在全部集成时应当满足需求；
- 编制设计文档以便编码者和以后的产品维护者可以理解；
- 在设计中包括足够的信息可支持计划、设计和执行测试；
- 控制设计配置，保证所有的文档都是完整的和可发布的，特别是使用混合媒体(例如，图表、文字说明)的情况下。

满足上述目标能够保证所有的需求在设计中得到体现，同时设计能够满足需求，并且是可测试的，能导致可测试的代码。验证和确认的计划者的责任是为设计的产品(包括中间阶段的规格)选择验证和确认任务，并确信在实际中已满足了这些目标。

验证和确认的计划者应选择适合每一个层次设计的带专有特性的验证和确认任务及其相应的方法。当在每个设计层的验证和确认任务重复时，所采用的验证和确认方法或技巧

应改变。例如，在概要的设计中对算法选择进行评估是合适的，在详细设计中算法进行数学分析更合适些。

验证和确认工作的范围由设计工作的复杂性来决定。当计划设计阶段的验证和确认任务时，应考虑以下方面：

- 项目计划所连带的责任(例如，支持关键设计的评审)；
- 设计方法学；
- 设计标准；
- 设计中的关键点和高难度部分；
- 需要证明的设计假设(或证明的引用)；
- 罗列需要进一步分析的复杂算法；
- 需要进行大小和性能分析的资源限制(如，可利用的计算机硬件，时间限制等)；
- 需要进行安全性分析的数据库的保密和访问；
- 设计的层次(如，不同的 V&V 任务和方法分别适用于概要设计和详细设计上)；
- 单元测试和集成测试所需的不同方法；
- 表达设计的媒体和格式。

验证和确认的计划表应提供注解和在设计验证和确认任务中标注异常情况。另外，潜在风险也应提出来，这些已标识的风险的更新列表应安排人维护，它们应与开发中的测试计划相对应。

该阶段包括的任务有：

- 可跟踪性分析；
- 软件设计评估；
- 接口分析；
- 危险程度分析；
- 构件测试计划 V&V；
- 接口测试计划 V&V；
- 测试设计 V&V；
- 冒险分析；
- 风险分析。

12.4.5 实现阶段的验证和确认

实现阶段是软件生命周期中从设计文档到调试形成软件产品的阶段。在实现阶段，验证和确认的任务应集中在代码和代码符合设计规格和编码标准的程度上，在这个阶段，验证和确认的目标是确定代码的质量。

代码的质量由几个方面来确定。设计规格可跟踪到程序相应的代码。代码可跟踪到设计需求。这些步骤的开展可确保没有需求要增加、修改或被遗漏。分析程序接口并与接口文档相对照。详细评估程序的评估工作应开展，以分析程序是否对设计说明做了正确翻译，是否与程序编码标准相符。实现阶段的另一个活动是对不同测试(如单元、集成、系统和验收)的测试用例和测试过程的生成。在单元层次上的测试过程将被执行。

该阶段包括的任务有：

- 可跟踪性分析；
- 源代码和源代码文档评估；
- 接口分析；
- 危险程度分析；
- 测试用例 V&V；
- 测试程序 V&V；
- 构件测试执行 V&V；
- 冒险分析；
- 风险分析。

12.4.6 测试阶段的验证和确认

这个测试阶段是软件生命周期中的一个阶段。在这个阶段，软件产品的单元被评估和集成，软件产品被评估以确定是否满足需求。在测试阶段，所有被计划的测试阶段的验证和确认活动都应该完成。因此，制定测试过程、执行测试和分析结果的计划高度地依赖于诸如组织、进度、资源、责任、工具、技术和方法论等因素。组织的性质和在组织内如何开展验证和确认任务，将决定测试阶段的活动成功与否。

整个软件开发的安排将决定测试阶段何时开始和测试结果大致需多久才能得出。这个安排表应定期地被评估以确保有充足的时间来完成测试阶段。作为一个实践准则，一些从业者建议，充足的时间包括每个测试进行三次的时间——第一次是测试本身的测试，第二次是源代码的测试，第三次是回归测试。

在这个阶段正确地协调和下达完成期限是重要的。大项目往往包括许多雇员和各种不同的组织。例如，开发组织或第三方供应商提供硬件或固件（硬件和软件的组合件）。另外，可能需要客户提供所需的信息、数据或软件。所有这些人员可能要同时但彼此独立地为这些复杂的相互关联的任务而工作。因为他们都有相互依赖关系，要求完成期限是必须的。通过诸如变更控制，配置管理和报告等控制机制来实现整体协同工作也是很重要的。

该阶段包括的任务有：

- 可跟踪性分析；
- 确认 V&V 测试过程的制定和校验；
- 集成 V&V 测试的执行和校验；
- 系统 V&V 测试的执行和校验；
- 确认 V&V 测试的执行和校验；
- 冒险分析；
- 风险分析。

12.4.7 安装和校验阶段的验证和确认

安装和校验阶段是软件生命周期中的一个阶段，在该阶段中软件产品被集成到其运行

环境中，并经过测试以保证软件满足运行要求。安装特性包括安装人员、安装过程持续时间、安装地点数目、系统版本数目和配置的齐全性。

安装过程将一个完整的、已通过测试的应用程序安装到运行环境中，该安装方式应当满足使用该系统时的用户要求。有时这个过程与开发工作是分离的，并且是由另外一个组织来完成，而该组织并非开发者或是测试者（举例来说：由现场或客户支持工程师来完成）。有时安装和校验过程是由软件的终端用户来完成。这种情况在个人计算机软件行业中很典型。在很多情况下，安装过程在很短的时间内就可完成，有时需要一小时或几个小时。这时，软件在其生产环境中应当开始运行。有时安装是分步进行的，初次先交付一个或多个子系统，每一次安装后由用户进行一段时间的确认测试。在提交进行确认测试前，每个子系统应当能够很好地融合到先前的子系统中。

安装可能会重复若干次，对每一个安装场地、每一个软件版本都要进行一次。根据每一个安装场地的不同，产品可能要进行相应的修改（举例来说，设置与场地相关的参数，建立与地点相关的命令表等）；每个场地的安装流程要求也可能不同。

验证和确认的安装和校验过程可以保证：

- 每一个与安装场地相关的设置正确、完整；
- 每一个场地的安装说明是准确的，并且完备地表现该设置；
- 安装说明的内容应当适合安装人员的水平；
- 面向用户的校验测试应足以说明安装是成功的；
- 用来发货安装的软件应与通过审核和批准的软件是同一个软件。

验证和确认的安装和校验的职责可以划分给开发部门和用户。用户在安装过程中起重要的作用。任何分配给用户的验证和确认任务都应当清晰地被定义并有文档说明。在预安装的验证和确认中应当为用户建立明确的安装流程。

由于以上列出的所有原因，制定验证和确认的安装校验进度计划可能很困难。假如软件要在很多场所安装，事先就有必要制定一个通用的安装进度计划表，再根据时间和资源的因素为每一个特殊的安装进行修改。一个过程流程图或检验清单将会非常有用。

制定 V&V 安装和校验计划时，需要考虑以下各点：

- 核实准确性和完整性。通过控制准确性和完整性，保证安装前、中、后的数据完整。例如：如果一个数据文件要被重新格式化，就设计一个测试，说明重新格式化时保护了文件的完整性。如果要维持整个控制过程，就应当把最终控制与初始控制进行核对。
- 保留并核实安装过程的检查跟踪报告。核实在安装过程中发生的流程和修改都应当有记录。
- 保证先前系统的完整性。很多情况下，被安装的软件是要替代一个已经存在的系统。要保证在安装过程中，在新系统被正式接受并声明可以运行之前，允许现存系统继续运行。很有必要让新旧两个系统并行运行一段时间，或保留老系统以防新系统失败。
- 核实是否遵循安装和校验标准。确保安装校验过程按照适当的标准、流程、指导方针执行。

在安装完成之前可获得任一安装测试结果是非常重要的。这种测试（或校验）的目的

就是确定安装是否成功。这通常表明在测试开始之前对测试结果应有所预料。

该阶段包括的任务有：

- 安装配置审计；
- 安装检验；
- 冒险分析；
- 风险分析；
- V&V 最终报告。

12.4.8 运行和维护阶段的验证和确认

运行和维护阶段是软件生命周期中的一段时间，在这段时间内，软件产品工作在它的运行环境之中，进行性能监视，如果有必要，为了纠正错误或满足变化了的需求还要进行修改。

有时运行和维护阶段并不像软件生命周期中的顺序子集，可能会有多个软件版本使用并同时支持。每一个版本可能会出现在多个场地，每一个都有它自己特殊的安装参数，设备驱动程序或其他代码。不同安装可能会在不同的时间进行升级版本。对软件产品、应用以及用户其他考虑因素的高度依赖性可能会使操作和维护阶段更复杂。

如果考虑到这个复杂性，那么为本阶段的验证和确认任务提供明确的指导将非常困难，它常常是先前描述过的任务的重复。作为代替，在此只为运行和维护阶段的验证和确认提供一个标准。

有两种情况要考虑：

- 开发时，软件经过验证和确认(可能在标准控制下)；
- 开发时，软件没有进行充分或正式的验证和确认。

制定操作和维护过程中的 V&V 计划应该考虑软件属于两种情况的哪一种。

在为开发初期做过验证和确认的软件做验证和确认计划就有很多有利因素。会存在有完整的、最新的文档。审核和批准过程可以适当地、有效地使用。最后，还会有历史记录作为下一步计划的基础。这些优点可以帮助验证和确认计划者弥补在维护过程中会遇到的常见困难。许多关键的开发人员可能已经离开了这个工程，也许少掉了一个核心维护班子，也许一个开发组织正式转移为一个维护组织，和原来组织没有联系。用户可能会根据他们在应用中的优先级同时报告问题和提出新功能。

在维护之前，对软件没有进行验证和确认，计划者就会面临以上的陷阱以及一些新的问题。文档可能不存在或已经失效(这样可能比根本没有文档更糟)。开发者和管理者可能不喜欢验证和确认。当验证和确认被第一次执行时，这里可能没有用于计划的基础。

关键性软件应该在维护中被验证和确认，不管它在开发过程中是否曾被验证和确认。如果开发过程中软件被验证过，在维护过程中验证和确认的投入将会产生重要的回报。如果软件先前没有被验证，验证和确认可以给以前的混乱带来一些秩序，那样可以使维护人员的工作好做一些。

在操作维护阶段不能再像其他阶段那样，没有完整的、适时的文档记录验证和确认过程。操作和维护过程中的很大一部分工作会花费在准备和更新那些不合适的文档中。使用

残缺或不合适的文档进行软件验证和确认计划时,管理者、开发者和用户会感觉到验证和确认工作的效力严重打折扣。如果要拓展新的开发或要彻底掌握现存系统,完善残缺的文档是非常有价值的。

该阶段包括的任务有:

- 评估新的约束和限制;
- 变更建议评议;
- 操作程序评估;
- SVVP 修订;
- 异常评估;
- 危险程度分析;
- 移植评议;
- 更换评议;
- 冒险分析;
- 风险分析;
- 任务反复。

12.4.9 验证和确认任务总结

表 12-3 给出了所有验证和确认过程中任务的一个概述,其中包括输入、任务和输出。该表是 IEEE 标准提供的,其中包括了在上面没有提到的获取过程和供应过程的一些活动。表 12-4、表 12-5、表 12-6 给出了各阶段的最小任务选择,在这些表中,使用了表 12-2 定义的软件集成级别。

表 12-3 验证和确认的任务、输入和输出汇总表

V&V 任务	要求的输入	要求的输出
V&V 管理活动(与其他活动并行)		
(1) 生成软件确认和验证计划 (SVVP)	<ul style="list-style-type: none"> ✓ SVVP ✓ 摘要 ✓ 概念文档 ✓ 供应者开发计划 	<ul style="list-style-type: none"> ✓ 更新后的 SVVP
(2) 基线变更评审	<ul style="list-style-type: none"> ✓ SVVP ✓ 建议变更 ✓ 冒险分析报告 ✓ 基于 SVVP 的风险标识 	<ul style="list-style-type: none"> ✓ 更新后的 SVVP ✓ 异常报告
(3) V&V 管理评审	<ul style="list-style-type: none"> ✓ SVVP ✓ 供应者开发计划 ✓ V&V 任务结果 	<ul style="list-style-type: none"> ✓ 更新后的 SVVP
(4) 管理和技术评审支持	<ul style="list-style-type: none"> ✓ V&V 任务结果 ✓ 评审材料 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告

续表

V&V 任务	要求的输入	要求的输出
(5) 与其他流程的接口	✓ SVVP ✓ 来源于其他流程的数据在 SVVP 中被标识	✓ 更新后的 SVVP
获取支持 V&V 活动(获取过程)		
(1) 界定 SVVP 投入	✓ 初步的系统描述 ✓ 需求陈述 ✓ 系统集成级别方案	✓ 更新后的 SVVP
(2) 计划 SVVP 投入与供应者之间的接口	✓ SVVP ✓ RFP ✓ 合同 ✓ 支持者开发计划和进度	✓ 更新后的 SVVP
(3) 系统需求评审	✓ 初步的系统描述 ✓ 需求陈述 ✓ 用户需求 ✓ RFP	✓ 任务报告 ✓ 异常报告
计划 V&V 活动(供应过程)		
(1) 计划 V&V 与供应者之间的接口	✓ SVVP ✓ 合同 ✓ 供应者开发计划	✓ 更新后的 SVVP
(2) 合同验证	✓ SVVP ✓ RFP ✓ 合同 ✓ 用户需求 ✓ 供应者开发计划	✓ 更新后的 SVVP 任务报告 ✓ 异常报告
概念 V&V 活动(开发过程)		
(1) 概念文档评估	✓ 概念文档 ✓ 供应者开发计划 ✓ 用户需求 ✓ 获取需求	✓ 任务报告 ✓ 异常报告
(2) 关键程度分析	✓ 概念文档 ✓ 开发集成度分配	✓ 任务报告 ✓ 异常报告
(3) 硬件/软件/用户需求分配	✓ 用户需求 ✓ 概念文档	✓ 任务报告 ✓ 异常报告
(4) 可跟踪性分析	✓ 概念文档	✓ 任务报告 ✓ 异常报告
(5) 冒险分析	✓ 概念文档	✓ 任务报告 ✓ 异常报告
(6) 风险分析	✓ 概念文档 ✓ 供应者开发计划 ✓ 冒险分析报告 ✓ V&V 活动结果	✓ 任务报告 ✓ 异常报告

续表

V&V 任务	要求的输入	要求的输出
需求 V&V 活动(开发过程)		
(1)可跟踪性分析	✓ 概念文档 ✓ SRS ✓ IRS	✓ 任务报告 ✓ 异常报告
(2)软件需求评估	✓ 概念文档 ✓ SRS ✓ IRS	✓ 任务报告 ✓ 异常报告
(3)接口分析	✓ 概念文档 ✓ SRS ✓ IRS	✓ 任务报告
(4)关键性分析	✓ 任务报告 ✓ SRS ✓ IRS	✓ 任务报告 ✓ 异常报告
(5)系统 V&V 测试计划制定和验证	✓ 概念文档 ✓ SRS ✓ IRS ✓ 用户文档 ✓ 系统测试计划	✓ 异常报告 ✓ 系统 V&V 测试计划
(6)确认 V&V 测试计划制定和验证	✓ 概念文档 ✓ SRS ✓ IRS ✓ 用户文档 ✓ 确认测试计划	✓ 确认 V&V 测试计划 ✓ 异常报告
(7)配置管理评价	✓ 软件配置管理过程文档	✓ 任务报告 ✓ 异常报告
(8)危险分析	✓ SRS ✓ IRS ✓ 危险分析报告	✓ 任务报告 ✓ 异常报告
(9)风险分析	✓ 概念文档 ✓ SRS ✓ IRS ✓ 供应者开发计划 ✓ 冒险分析报告 ✓ V&V 任务结果	✓ 任务报告 ✓ 异常报告
设计 V&V 活动(开发过程)		
(1)可跟踪性分析	✓ SRS ✓ SDD ✓ IRS ✓ IDD	✓ 任务报告 ✓ 异常报告

续表

V&V 任务	要求的输入	要求的输出
(2) 软件设计评估	<ul style="list-style-type: none"> ✓ SRS ✓ IRS ✓ SDD ✓ IDD ✓ 设计标准 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(3) 接口分析	<ul style="list-style-type: none"> ✓ 概念文档 ✓ SRS ✓ IRS ✓ SDD ✓ IDD 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(4) 关键性分析	<ul style="list-style-type: none"> ✓ 任务报告 ✓ SDD ✓ IDD 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常分析
(5) 构件 V&V 测试计划制定和验证	<ul style="list-style-type: none"> ✓ SRS ✓ SDD ✓ IRS ✓ IDD ✓ 组件测试计划 	<ul style="list-style-type: none"> ✓ 组件 V&V 测试计划 ✓ 异常报告
(6) 集成 V&V 测试计划制定和验证	<ul style="list-style-type: none"> ✓ SRS ✓ IRS ✓ SDD ✓ IDD ✓ 集成测试计划 	<ul style="list-style-type: none"> ✓ 集成 V&V 测试计划 ✓ 异常报告
(7) V&V 测试设计制定和验证	<ul style="list-style-type: none"> ✓ SDD ✓ IDD ✓ 用户文档 ✓ 测试计划 ✓ 测试设计 	<ul style="list-style-type: none"> ✓ 组件 V&V 测试设计 ✓ 集成 V&V 测试设计 ✓ 系统 V&V 测试设计 ✓ 确认 V&V 测试设计 ✓ 异常报告
(8) 危险分析	<ul style="list-style-type: none"> ✓ SDD ✓ IDD ✓ 冒险分析报告 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(9) 风险分析	<ul style="list-style-type: none"> ✓ SDD ✓ IDD ✓ 供应者开发计划 ✓ V&V 任务结果 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
实现 V&V 活动(开发过程)		
(1) 可跟踪性分析	<ul style="list-style-type: none"> ✓ SDD ✓ IDD ✓ 源代码 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常分析
(2) 源代码和源代码文档评估	<ul style="list-style-type: none"> ✓ 源代码 ✓ SDD ✓ IDD ✓ 编码标准 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告

续表

V&V 任务	要求的输入	要求的输出
(3) 接口分析	<ul style="list-style-type: none"> ✓ 概念文档 ✓ SDD ✓ IDD ✓ 源代码 ✓ 用户文档 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(4) 关键性分析	<ul style="list-style-type: none"> ✓ 任务报告 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(5) V&V 测试用例制定和验证	<ul style="list-style-type: none"> ✓ SRS ✓ IRS ✓ SDD ✓ IDD ✓ 用户文档 ✓ 测试设计 ✓ 测试用例 	<ul style="list-style-type: none"> ✓ 组件 V&V 测试用例 ✓ 集成 V&V 测试用例 ✓ 系统 V&V 测试用例 ✓ 确认 V&V 测试用例 ✓ 异常报告
(6) V&V 测试过程制定和验证	<ul style="list-style-type: none"> ✓ SRS ✓ IRS ✓ SDD ✓ IDD ✓ 用户文档 ✓ 测试用例 ✓ 测试过程 	<ul style="list-style-type: none"> ✓ 组件 V&V 测试过程 ✓ 集成 V&V 测试过程 ✓ 系统 V&V 测试过程 ✓ 异常报告
(7) 组件 V&V 测试执行和验证	<ul style="list-style-type: none"> ✓ 源代码 ✓ 可执行代码 ✓ SDD ✓ IDD ✓ 组件测试计划 ✓ 组件测试过程 ✓ 组件测试结果 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(8) 冒险分析	<ul style="list-style-type: none"> ✓ 源代码 ✓ SDD ✓ IDD ✓ 冒险分析报告 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(9) 风险分析	<ul style="list-style-type: none"> ✓ 源代码 ✓ 供应者开发计划 ✓ 冒险分析报告 ✓ V&V 任务结果 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
测试 V&V 活动(开发过程)		
(1) 可跟踪性分析	<ul style="list-style-type: none"> ✓ V&V 测试计划 ✓ V&V 测试设计 ✓ V&V 测试过程 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告

续表

V&V 任务	要求的输入	要求的输出
(2) 确认 V&V 测试过程制定和验证	<ul style="list-style-type: none"> ✓ SDD ✓ IDD ✓ 源代码 ✓ 用户文档 ✓ 确认测试计划 ✓ 确认测试过程 	<ul style="list-style-type: none"> ✓ 确认 V&V 测试过程 ✓ 异常报告
(3) 集成 V&V 测试执行和验证	<ul style="list-style-type: none"> ✓ 源代码 ✓ 可执行代码 ✓ 集成测试计划 ✓ 集成测试过程 ✓ 集成测试结果 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(4) 系统 V&V 测试执行和验证	<ul style="list-style-type: none"> ✓ 源代码 ✓ 可执行代码 ✓ 系统测试计划 ✓ 系统测试过程 ✓ 系统测试结果 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(5) 确认 V&V 测试执行和验证	<ul style="list-style-type: none"> ✓ 源代码 ✓ 可执行代码 ✓ 用户文档 ✓ 确认测试计划 ✓ 确认测试过程 ✓ 确认测试结果 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(6) 冒险分析	<ul style="list-style-type: none"> ✓ 源代码 ✓ 可执行代码 ✓ 测试结果 ✓ 冒险分析报告 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(7) 风险分析	<ul style="list-style-type: none"> ✓ 供应者开发计划 ✓ 冒险分析报告 ✓ V&V 任务结果 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
安装和检验 V&V 活动(开发过程)		
(1) 安装配置审计	<ul style="list-style-type: none"> ✓ 安装包 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(2) 安装检验	<ul style="list-style-type: none"> ✓ 用户文档 ✓ 安装包 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(3) 冒险分析	<ul style="list-style-type: none"> ✓ 安装包 ✓ 冒险分析报告 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(4) 风险分析	<ul style="list-style-type: none"> ✓ 安装包 ✓ 供应者开发计划 ✓ V&V 任务结果 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(5) V&V 最终报告	<ul style="list-style-type: none"> ✓ V&V 活动总结报告 	<ul style="list-style-type: none"> ✓ V&V 最终报告
操作 V&V 活动(操作阶段)		
(1) 新约束评估	<ul style="list-style-type: none"> ✓ SVVP ✓ 新约束 	<ul style="list-style-type: none"> ✓ 任务报告

续表

V&V 任务	要求的输入	要求的输出
(2) 建议变更评价	<ul style="list-style-type: none"> ✓ 建议变更 ✓ 安装包 	<ul style="list-style-type: none"> ✓ 任务报告
(3) 操作过程评估	<ul style="list-style-type: none"> ✓ 操作过程 ✓ 用户文档 ✓ 概念文档 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(4) 冒险分析	<ul style="list-style-type: none"> ✓ 操作过程 ✓ 冒险分析报告 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(5) 风险分析	<ul style="list-style-type: none"> ✓ 安装包 ✓ 建议变更 ✓ 冒险分析报告 ✓ 供应者开发计划 ✓ 操作问题报告 ✓ V&V 任务结果 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
维护 V&V 活动(维护过程)		
(1) 修订 SVVP	<ul style="list-style-type: none"> ✓ SVVP ✓ 被认可的变更 ✓ 安装包 ✓ 供应者开发计划 	<ul style="list-style-type: none"> ✓ 更新后的 SVVP
(2) 建议变更评估	<ul style="list-style-type: none"> ✓ 建议变更 ✓ 安装包 ✓ 供应者开发计划 	<ul style="list-style-type: none"> ✓ 任务报告
(3) 异常评估	<ul style="list-style-type: none"> ✓ 异常报告 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(4) 关键性分析	<ul style="list-style-type: none"> ✓ 建议变更 ✓ 安装包 ✓ 完整性级别维护 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(5) 移植评估	<ul style="list-style-type: none"> ✓ 安装包 ✓ 被认可的变更 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(6) 更换评估	<ul style="list-style-type: none"> ✓ 安装包 ✓ 被认可的变更 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常评估
(7) 冒险分析	<ul style="list-style-type: none"> ✓ 建议变更 ✓ 安装包 ✓ 冒险分析报告 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告
(8) 风险分析	<ul style="list-style-type: none"> ✓ 安装包 ✓ 建议变更 ✓ 冒险分析报告 ✓ 供应者开发计划 ✓ 操作问题报告 ✓ V&V 任务结果 	<ul style="list-style-type: none"> ✓ 任务报告
(9) 任务重复	<ul style="list-style-type: none"> ✓ 被认可的变更 ✓ 安装包 	<ul style="list-style-type: none"> ✓ 任务报告 ✓ 异常报告

表 12-4 获取和供应阶段最小 V&V 任务集

生命周期过程	获取				供应			
V&V 活动	获取供应 V&V 活动				计划 V&V 活动			
软件完整性级别	Level				Level			
	4	3	2	1	4	3	2	1
确认 V&V 执行和验证								
确认 V&V 测试计划制定和验证								
异常评估								
组件 V&V 执行和验证								
概念文档评估								
配置管理评估								
合同验证					*			
关键性分析								
新约束评估								
硬件/软件/用户需求分配分析								
冒险分析								
安装检验								
安装配置审计								
接口分析								
集成 V&V 执行和验证								
V&V 管理								
a) 基线变更评估								
b) 与其他流程的接口	*	*			*	*		
c) 管理和技术评审支持								
d) V&V 管理评审	*	*	*		*	*	*	
e) SVVP 计划制定								
移植评估								
操作过程评估								
计划与其他流程间的接口	*	*	*		*	*	*	
建议变更评估								
风险分析								
更换评估								

续表

生命周期过程	获取				供应			
V&V 活动	获取供应 V&V 活动				计划 V&V 活动			
软件完整性级别	Level				Level			
界定 V&V 投入	*	*	*					
软件设计评估								
软件需求评估	*	*	*	*				
SVVP 修订								
源代码和源代码文档评估								
软件需求评审								
系统 V&V 测试执行和验证								
系统 V&V 测试计划制定和验证								
任务重复								
可跟踪性分析								
V&V 最终报告								
V&V 测试设计制定和验证								
a) 组件								
b) 集成								
c) 系统								
d) 确认								
V&V 测试用例制定和验证								
a) 组件								
b) 集成								
c) 系统								
d) 确认								
V&V 测试过程制定和验证								
a) 组件								
b) 集成								
c) 系统								

表 12-5 开发阶段最小 V&V 任务集

生命周期过程	开 发																							
V&V 活动	概念 V&V 活动				需求 V&V 活动				设计 V&V 活动				实现 V&V 活动				测试 V&V 活动				安装/检验 V&V 活动			
软件完整性级别	Level				Level				Level				Level				Level				Level			
	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1
确认 V&V 执行和验证																	*	*	*					
确认 V&V 测试计划制定和验证					*	*	*																	
确认测试过程确定制定和验证																	*	*	*					
异常评估																								
组件 V&V 执行和验证													*	*	*									
组件测试计划制定和评估									*	*	*													
概念文档评估	*	*	*																					
配置管理评估					*	*																		
合同验证																								
关键性分析	*	*	*																					
新约束评估																								
硬件/软件/用户需求分配分析	*																							
冒险分析	*	*			*	*			*	*			*	*			*	*			*	*		
安装检验																					*	*		
安装配置审计																					*	*		
接口分析					*	*	*		*	*	*		*	*	*									
集成 V&V 执行和验证																	*	*	*					
V&V 管理																								
a) 基线变更评估	*				*	*	*		*	*	*		*	*	*		*	*	*		*	*	*	
b) 与其他流程的接口					*	*			*	*			*	*			*	*			*	*		
c) 管理和技术评审支持					*	*			*	*			*	*			*	*			*	*		

续表

生命周期过程	开 发																							
V&V 活动	概念 V&V 活动				需求 V&V 活动				设计 V&V 活动				实现 V&V 活动				测试 V&V 活动				安装/检验 V&V 活动			
软件完整性级别	Level				Level				Level				Level				Level				Level			
	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1
V&V 测试用例制定和验证																								
a) 组件									*	*	*													
b) 集成									*	*	*	*												
c) 系统									*	*	*	*												
d) 确认									*	*	*													
V&V 测试过程制定和验证																								
a) 组件									*	*	*													
b) 集成									*	*	*	*												
c) 系统									*	*	*	*												

表 12-6 操作和维护阶段最小 V&V 任务集

生命周期过程	获 取				供 应			
V&V 活动	获取供应 V&V 活动				计划 V&V 活动			
软件完整性级别	Level				Level			
	4	3	2	1	4	3	2	1
确认 V&V 执行和验证								
确认 V&V 测试计划制定和验证								
异常评估					*	*	*	
组件 V&V 执行和验证								
概念文档评估								
配置管理评估								
合同验证								
关键性分析					*	*	*	
新约束评估	*	*	*					
硬件/软件/用户需求分配分析								
冒险分析	*	*			*	*		
安装检验								

续表

生命周期过程	获 取				供 应			
V&V 活动	获取供应 V&V 活动				计划 V&V 活动			
软件完整性级别	Level				Level			
	4	3	2	1	4	3	2	1
安装配置审计								
接口分析								
集成 V&V 执行和验证								
V&V 管理								
a) 基线变更评估	*	*			*	*		
b) 与其他流程的接口	*	*			*	*		
c) 管理和技术评审支持	*	*			*	*		
d) V&V 管理评审	*	*	*	*	*	*	*	*
e) SVVP 计划制定					*	*	*	*
移植评估					*	*		
操作过程评估	*	*						
计划与其他流程间的接口								
建议变更评估	*	*	*		*	*	*	
风险分析	*	*			*	*		
更换评估					*	*		
界定 V&V 投入								
软件设计评估								
软件需求评估								
SVVP 修订					*	*	*	*
源代码和源代码文档评估								
软件需求评审								
系统 V&V 测试执行和验证								
系统 V&V 测试计划制定和验证								
任务重复					*	*	*	*
可跟踪性分析								
V&V 最终报告								
V&V 测试设计制定和验证								
a) 组件								

续表

生命周期过程	获 取				供 应			
V&V 活动	获取供应 V&V 活动				计划 V&V 活动			
软件完整性级别	Level				Level			
	4	3	2	1	4	3	2	1
b) 集成								
c) 系统								
d) 确认								
V&V 测试用例制定和验证								
a) 组件								
b) 集成								
c) 系统								
d) 确认								
V&V 测试过程制定和验证								
a) 组件								
b) 集成								
c) 系统								

12.5 验证和确认的报告

验证和确认报告记录了验证和确认执行的状态以及在验证和确认执行中发现的所有需要关注的内容。许多不同的人和组织承担验证和确认任务,验证和确认活动结果也会涉及各种各样的人员。读者的多样化要求对报告的发行、格式和内容进行合适的修改。验证和确认报告可以加深对软件产品开发过程的认识,加强对产品更完整的理解,但是只能对验证和确认过程的参与者共享验证和确认信息。

对错误的响应时间会极大地影响因软件错误而引起的成本花费,响应时间越短,成本越小。响应时间就是软件中引入一个错误直到发现并修改错误之间的这段时间。在软件开发中并行执行验证和确认的最大好处来自于将验证和确认发现的任何错误及时地通知开发者。

所有的验证和确认活动都要被报告。进行验证和确认规划时要给报告安排足够的时间,虽然明确的报告需求会随着活动而改变。验证和确认报告具有多种命名和格式,比如备忘录、简介、做过标记的历史文档备份、会议记录、工作项、状态报告、评估分析报告、审计报告、检查报告、测试报告、异常报告、失败报告,等等。所有这些报告都可以纳入标准或者可选范畴。

12.5.1 标准要求的报告

IEEE 要求提供4种类型的验证和确认报告, 它们的等级范围涉及从软件中发现个别明显的缺点(异常报告)到对这个开发工程的复杂的评估(最终验证和确认报告)。验证和确认要求的4种类型报告如下:

- 异常报告。这些报告要及时反映给开发者。突然的改变要求建立流程来记录和描述发现的异常, 然后提供相关的信息给开发者。异常报告的详细描述包括对每一个异常的描述、每个异常准确的范围的描述、重现的技巧的描述和对异常要求的管理的描述。
- 任务报告。这些报告总是跟随着每一个严格定义的验证和确认任务。任务报告会被验证和确认的工程管理者查看, 以作为验证和确认不断改善的证据。任务报告还为开发或验证确认计划中间过程的修正提供关键的需求点, 并被纳入验证和确认管理历史。验证和确认相关任务的效率和困难可能导致战术上的调整, 比如对方法和工具的选择。更进一步, 现行的实际的任务结果也可以促使战略上的改变, 包括验证和确认资源的再分配或进度表的修改。任务报告的内容和进度, 应该定位在将别的任务的输出作为自己输入的需求上(例如: 当测试计划需要跟踪分析的结果时), 并且是随着验证和确认任务的顺序和时间安排而计划的。
- 阶段总结报告。这些报告总结与合并不同组织执行的每一个生命周期阶段的验证和确认活动结果。阶段总结报告可以作为一个广泛的正规文档或一个简要的非正式的通知, 这取决于在一个特定阶段验证和确认活动进行的深度和广度。这个报告也可以是一个阶段一个回顾的一个简要提交, 例如: 初步的设计回顾。如果信息要求非常严格而不能等到整个阶段的结尾, 就可以在合适的时候产生一个临时的报告。
- 最终报告。这个报告综合软件异常报告、验证和确认任务报告以及所有验证和确认的阶段报告的结果。验证和确认最终报告是在所有验证和确认任务完成时准备的, 它会提供软件质量的一个总的评估及对产品和/或开发过程的任何建议。

12.5.2 标准可选报告

IEEE 标准中还提到了其他一些可选报告, 这些可选报告也是一些特殊验证和确认的研究或其他未预料活动的结果。给出这些报告的惟一特性, 对它们的格式和内容没有特别的指导方针。和所有的验证和确认报告一样, 可选报告要求按时、适当地分发。它们要明确指出验证和确认活动的目的, 描述使用的方法, 并在一个合适的层次报告它们的结果。

12.6 本章小结

本章详细描述了软件验证和确认过程, 就是检查开发活动是否按规范进行、软件是否

满足需求规定和用户要求。检查手段可以是对软件产品和过程的分析、评审、审视、评估和测试。验证和确认过程把软件放在整个系统中进行评估,包括操作系统、硬件、接口软件、操作者和用户。

验证和确认是一个关于过程的标准,包含了所有的软件生命周期过程如:获取、支持、开发、操作和维护。该标准与所有生命周期模型兼容。当验证和确认过程与软件开发过程并行时显得尤其有效,否则将有可能达不到预期目标。本章中验证过程和确认过程是综合在一起讨论的,但是在某些情况下,这两个过程也可以分别考虑。

本章所描述的验证和确认过程可以被应用到任何新开发、维护和重用的“软件”中。这里的“软件”包括固件、微代码和文档。由于软件是影响系统行为和性能的重要组成部分,所以验证和确认过程必须考虑软件与其他部分的交互。

第 13 章 软件质量保证

一个软件产品的质量不是突然获得的。前面章节已经提到，只有通过全面的质量管理才能保证产品有一个好的质量。我们不能把质量寄希望于最后的测试阶段。很多产品经理常存在这么一个想法：测试最终会抓住产品所有的 Bug。这是一个非常危险的想法。事实早已证明，测试不能发现产品中所有的问题，而且 Bug 发现越迟，其发现成本越高，早期的 Bug 可能会导致后期的多个 Bug。因此，软件产品的质量是一个自始至终的过程，应当在每个阶段上得到保证。CMM 二级中定义的 KPA“软件质量保证”是一个保证过程质量的有效手段，本章将从概念和过程方面介绍这方面的知识。通过本章的学习，你需要了解以下内容：

1. 为什么要进行软件质量保证活动？
2. 软件质量保证活动的目标、承诺、能力、活动、度量和验证分别有哪些内容？
3. 软件质量保证活动的实施过程。

13.1 基本概念

软件质量保证(Software Quality Assurance SQA)是 CMM(Capability Maturity Model)二级的一个 KPA(Key Process Areas)。其正式的定义是：软件质量保证是一个系统性的活动，它为软件产品的可用性提供了保证。软件质量保证通过使用已建立的质量控制过程来保证软件的一致性，并延长软件的使用寿命^[8]。质量管理的一个基本准则是：如果没有跟踪，就等于什么也没做^[142]。在软件中，有如此之多的事情要做，管理者不可能跟踪每一件事情，而有效的软件管理需要精确地完成上千件事情，一些组织需要对此进行跟踪，这就是 SQA 角色的作用。

软件质量保证的目的是向管理者提供适当的对软件项目正使用的过程和正构造产品的可视性，包括评审和审计软件产品和开发活动以验证它们符合适用的规程和标准，给项目经理和其他有关经理提供这些评审和审计的结果^{[216][219]}。

SQA 组织不负责生成高质量的产品，也不负责制定质量计划，这些都是程序员们的职责。SQA 负责审计产品线的质量活动并就任何偏差向管理者提出警告。

为了有效地工作，SQA 需要和程序员们一起工作。SQA 需要了解计划、验证执行、监控每个任务的执行情况。但是，如果程序员们以 SQA 为敌，则 SQA 很难有效地工作。关键是 SQA 的协作和支持态度。如果 SQA 做事武断，对程序员们充满敌意，或者吹毛求疵，那么不管管理者给他们多大的支持，他们都很难有效地工作。

附录 E 的^[216]项从目标、执行的承诺、执行的能力、执行的活动、度量分析和验证实

现这 6 个方面对 SQA 这个 KPA 进行了说明。

13.1.1 目标

目标 1 软件质量保证活动必须是有计划的。

目标 2 软件产品和活动对适用的标准、规程和需求的遵从情况得到客观的验证。

目标 3 受影响的组和个人被通知了软件质量保证的活动和结果。

目标 4 高级管理者对软件项目内部不能解决的不一致问题进行了说明。

13.1.2 执行的承诺

承诺 1 项目遵循书面的实施软件质量保证(SQA)的组织方针。这个方针主要指:

(1) 对全部软件项目, SQA 功能到位。

(2) SQA 有一个向高级管理者报告的渠道, 它独立于:

- 项目经理;
- 项目的软件工程部;
- 其他的相关软件组, 例如有软件配置管理组, 文档支持组等。

组织必须确定一种组织机构, 它在组织的战略经营目标和经营环境上, 支持那些要求独立性的活动, 例如 SQA。独立性应该:

- 给担当 SQA 角色的个人提供组织上的自由度, 使他们成为高级管理者在软件项目上的“耳目”。
- 使得担当 SQA 角色的个人免受他们正在评审的软件项目的管理者所做的性能评价的影响。
- 使高级管理者相信正在报告的有关项目过程和产品的信息是客观的。

(3) 高级管理者定期地评审 SQA 活动和结果。

13.1.3 执行的能力

能力 1 存在一个用于项目的协调和实现 SQA 的组织, 例如 SQA 组。这个组是负责一系列任务或活动的部门、经理和个人的集合。组的规模可以变化, 从一个受指派的非全日制的单个人, 到几个从不同部门指派来的非全日制的个人, 再到几个全日制的个人。

建立这个组时应考虑的因素包括指派的任务和活动、项目的规模、组织机构和组织文化。某些组, 例如软件质量保证组, 关注于项目活动, 而其他组, 例如软件工程过程组, 则关注于整个组织的活动。

能力 2 充足的资源和资金被提供用于执行 SQA 活动, 包括:

- 指派一个经理专门负责项目的 SQA 活动。
- 指派一个在 SQA 任务方面是博学的, 并有权力采取适当的监督行动的高级经理接触和处理软件不符合问题。

在 SQA 向高级经理报告链上的全部经理均是在 SQA 的任务、责任和权力方面有

见识的。

- 有支持 SQA 活动的合适工具。支持工具的例子有：
 - ✓ 工作站；
 - ✓ 数据库程序；
 - ✓ 电子表格程序；
 - ✓ 审计工具。

能力3 SQA 组的成员被培训以便执行他们的 SQA 活动。培训的例子有：

- 软件工程技巧和实践
- 软件工程组和其他软件相关组的岗位任务及职责
- 用于软件项目的标准、规程和方法
- 软件项目的应用领域
- SQA 的对象、规程和方法
- SQA 组对软件活动的参与
- SQA 方法和工具的有效使用
- 人员间的交流

能力4 软件项目的成员接受有关 SQA 组的任务、职责、权力和价值等方面的定位。

13.1.4 执行的活动

活动1 根据文档化的过程为软件项目准备一份 SQA 计划。这个过程主要指的是：

- SQA 计划的制定是在整个项目计划的早期阶段，并平行于整个项目计划。
- 受影响的组和个人评审该 SQA 计划。

受影响的组及个人包括：

- ✓ 项目软件经理；
 - ✓ 其他软件经理；
 - ✓ 项目经理；
 - ✓ 顾客的 SQA 代表；
 - ✓ SQA 组对其报告不符合问题的高级经理；
 - ✓ 软件工程组(包括全部小组，诸如软件设计小组及软件任务领导组)。
- 对 SQA 计划进行管理和控制。

“进行管理和控制”意味着在给定时间(过去或现在)使用的工作产品的版本是已知的(即版本控制)，而且以受控的方式进行更改(即更改控制)。如果希望有此“进行管理和控制”所蕴含的更高程度的控制，则工作产品可置于配置管理的完整规则之下，正如在软件配置管理关键过程区域中所描述的。

活动2 按照 SQA 计划进行 SQA 组的活动。该计划包含：

- SQA 组的职责和权力。
- SQA 组的资源要求(包括职员、工具和设施)。
- 项目的 SQA 组活动的进度表和资金。
- SQA 组参加制定项目的软件开发计划、标准和规程的情况。

- SQA 组执行的评价。被评价的产品和活动包括：
 - ✓ 运行软件和支持软件；
 - ✓ 可交付的和不交付的产品；
 - ✓ 软件和非软件产品(例如文档)；
 - ✓ 产品开发和产品验证活动(例如运行测试用例)；
 - ✓ 生成产品时所从事的活动。
- 将由 SQA 组进行的审计和评审。
- 将用作 SQA 组评审和审计基础的项目标准和规程。
- 用于对不符合性问题建立文档和进行跟踪直至结束的规程。
- 这些规程可能作为计划的一部分而纳入，也可以通过索引那些包含它们的其他文档的方式而纳入。
- 要求 SQA 组生成的文档。
- SQA 活动给软件工程组和其他软件相关组提供反馈信息的方法和频率。

活动3 SQA 组参与准备和评审项目的软件开发计划、标准和规程。

- SQA 就以下几个方面对计划、标准和规程提供咨询和评审：
 - ✓ 对组织方针的符合性；
 - ✓ 对外部强加的标准和要求的符合性(例如工作陈述所要求的标准)；
 - ✓ 适合项目使用的标准；
 - ✓ 在软件开发计划中应阐述的专题；
 - ✓ 项目指定的其他领域。
- SQA 组验证计划、标准和规程已到位并可用于评审与审计软件项目。

活动4 SQA 组评审软件工程活动以验证符合性。

- 对照软件开发计划和指定的软件标准和规程去评价活动。
参考其他关键过程区域中的验证实施共同特点以便找到包括由 SQA 组进行特定评审和审计的实践。
- 对偏差进行鉴别和建立文档，并跟踪到结束。
- 验证纠正措施。

活动5 SQA 组审计指定的软件工作产品以验证符合性。

- 在交付给顾客之前，评价可交付的软件产品。
- 对照指定的软件标准、规程和合同要求评价软件工作产品。
- 对偏差进行鉴别和建立文档，并跟踪到结束。
- 验证纠正措施。

活动6 SQA 组定期向软件工程组报告其活动的结果。

活动7 按照已文档化的规程对在软件活动和软件工作产品中所鉴别出的偏差建立文档并加以处理。

该规程一般规定：

- 将不符合软件开发计划和指定的项目标准及规程的问题写成文档，并在可能处与适当的软件作业领导、软件经理或项目经理一起，加以解决。
- 有些不符合软件开发计划和指定的标准及规程的问题不能与软件作业领导、软件

经理或项目经理一起加以解决,将这些不符合问题写成文档并提交给指定的接收不符合问题的高级经理。

- 定期评审提交给高级经理的不符合问题直至解决它们为止。
- 对不符合问题的文档进行管理和控制。

活动8 当合适时, SQA 组与顾客的 SQA 人员一起对它的活动和发现进行定期评审。

13.1.5 度量分析

度量1 进行度量并将度量结果用于确定 SQA 活动的成本和进度状态。度量的例子有:

- SQA 活动的里程碑的完成情况与计划比较;
- 在 SQA 活动中所完成的工作、所花费的工作量和消耗的资金与计划做比较;
- 产品审计和活动评审的次数与计划相比较。

13.1.6 验证实现

验证1 高级管理者定期参与评审 SQA 活动。高级管理者定期评审的主要目的是在合适的抽象层次上并以及时的方式了解和洞察软件过程活动。评审间隔应该满足组织的需要,如果已存在报告例外情况的合适机制,那么评审间隔可以延长。参考软件项目跟踪和监督关键过程区域的验证1以便找到包括高级管理者监督评审的典型内容的实践。

验证2 项目经理既定期地也事件驱动地参与评审 SQA 活动。参考软件项目跟踪和监督关键过程区域的验证2以便找到包括项目管理者监督评审的典型内容的实践。

验证3 独立于 SQA 组的专家定期评审项目 SQA 组的活动和软件工作产品。

13.2 SQA 实施过程

CMM 给出了实施 SQA 需要达到的要求,但是并没有详细描述如何去实施 SQA,美国空海军战争系统中心(Space and Naval Warfare Systems Center)给出了关于实施 SQA 的过程^[217],该过程包括了8个步骤:

- 建立 SQA 组织;
- 选择 SQA 任务;
- 产生/维护 SQA 计划;
- 实施 SQA 计划;
- 产生/维护 SQA 规程;
- 标识 SQA 培训;
- 标识/选择 SQA 工具;
- 改进项目 SQA 过程。

这个过程具体可以参考图 13-1。

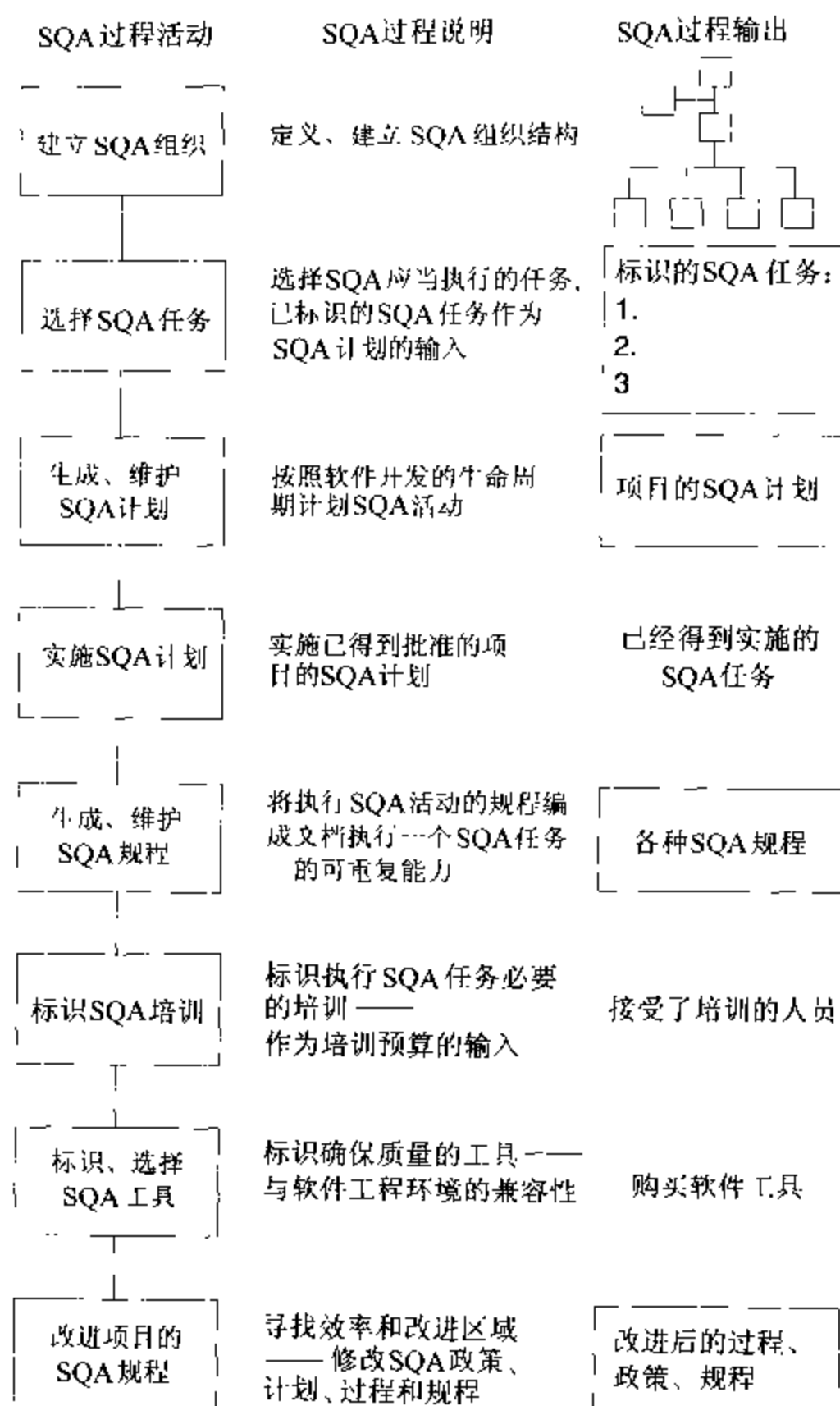


图 13-1 SQA 过程概述

13.2.1 建立SQA组织

许多好的软件实践要求对SQA组的独立性进行测量。这种独立性为SQA人员提供了一个关键的支持力量：也就是，如果产品质量受到危害时，SQA人员可以自由地把这种可能性直接向项目的上级报告。尽管实际中这种情况很少发生（因为几乎所有的问题在项目级得到了正确的解决），SQA组可以向项目上级报告这个事实使得SQA人员有能力保持这些问题中的大多数问题在项目级得到解决。

SQA功能可以分派给一个人或一个组，也可以分配给多个人或多个组。如果项目已经建立了一个独立的验证和确认（Independent Verification and Validation, IV&V）功能，那么SQA组可能已经在IV&V信息报告线上。

图 13-2 展示了 SQA 组织与项目组织之间关系的一个例子。项目的 SQA 组织, 包括其角色和责任, 应当被文档化到项目的 SQAP 中。在 SQAP、SDP、SCMP, 以及其他控制项目执行的计划中要求的组织结构图应保持一致。

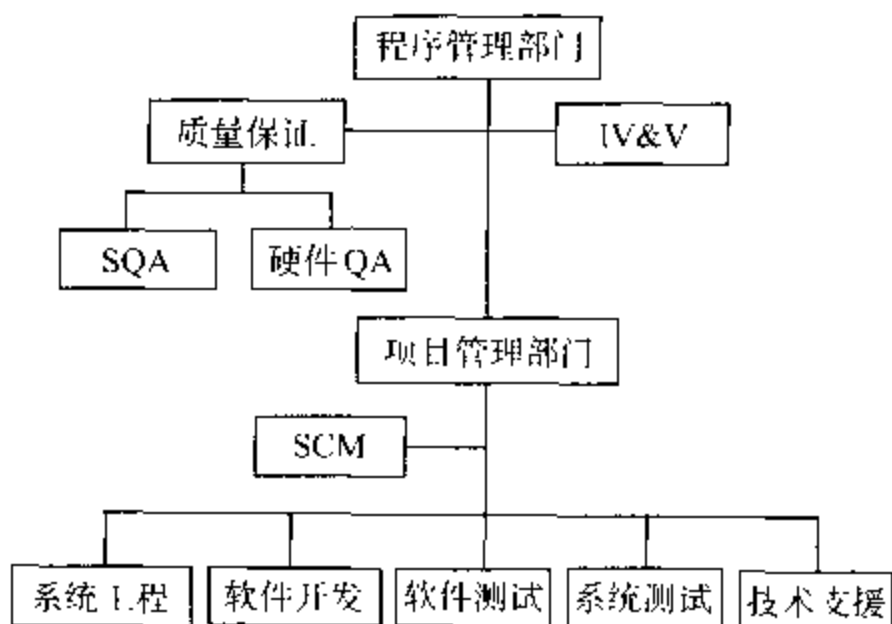


图 13-2 SQA 组织例子

13.2.2 选择 SQA 任务

在实施一个质量程序的过程中, 程序和管理应当包括 SQA 活动以确保定义在软件开发计划(SDP)中的软件开发过程或方法得到遵守。SQA 的角色应当:

- 标识并帮助减轻项目风险;
- 向高级管理者提供开发活动的可视性;
- 在软件开发过程中提供持续改进有效性方面的反馈。

这个过程活动需要有指定的人员来执行 SQA 和质量程序或项目管理任务以确认和计划要执行的 SQA 任务。其目标是选择那些能够为项目提供一个适当级别的 SQA 任务。任何 SQA 任务的标识, 要求充分考虑项目的软件开发计划(SDP)、软件配置管理计划(SCMP)以及其他控制项目执行的计划, 并与这些计划兼容。SQA 任务应当写在项目的软件质量保证计划(SQAP)文档中。

图 13-3 提供了 SQA 过程活动“选择 SQA 任务”的一个概述。该项目 SQA 任务的列表并未包括所有的 SQA 任务。

1. 软件产品、工具和设施的 SQA

软件产品的 SQA 评审就是依照规定的标准或指南来评价软件产品, 并报告错误、遗漏、以及用于纠正活动的注解。项目使用工具的 SQA 评估确保项目使用了适当的技术和工具来开发软件, 并且使用工具的项目组人员都接受了适当的培训。项目设施的 SQA 评估可以帮助管理者考虑或决择诸如将来购买哪些设备、扩大实验空间, 或者尽可能与其他项目共享设备资源等事情。

该项任务可以进一步被分解成下列更细的任务:

任务 1 评审软件产品。软件产品的 SQA 评审应当被应用到但不限于以下的软件产品: 开发计划、标准、规程、工具和过程; 软件需求; 软件设计; 代码; 跟踪设计、代码、

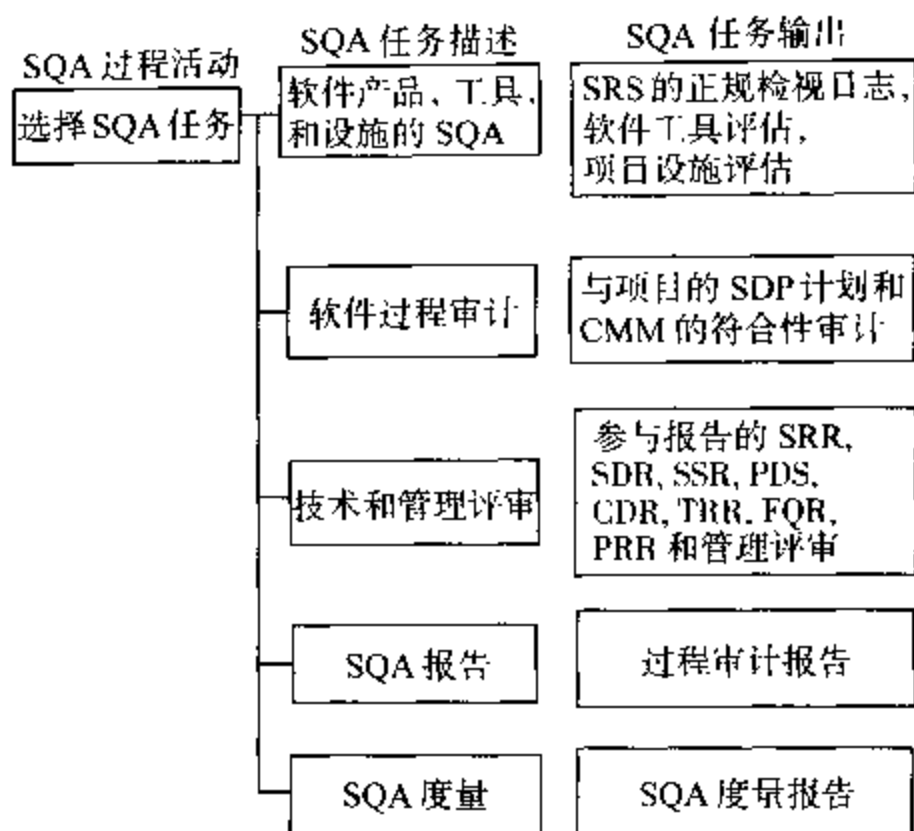


图 13-3 选择 SQA 任务的概述

数据库、手册和测试信息(软件单元测试、软件集成测试、软件性能测试)的控制机制。软件产品评审的方法可以使用非正规检视(也可以称为非作者评审或技术评审)和正规检视,这方面内容将在本书第 16 章进行详细讲解。

任务 2 评价软件工具。SQA 应当对用于软件开发和支持的,包括已有的和计划使用的工具进行评估。通过评估它们是否确实能完成其被设想的功能来对软件工具的充分性进行评估;通过评估工具的能力是否满足软件开发或软件支持的需要来评估其适用性。对要计划使用工具的可行性评估可以通过评估这些工具是否可以应用目前可用的或可获得的技术和计算机资源进行开发来进行。表 13-1 提供了一个报告软件工具评估结果的样本格式。执行这个任务的度量也应当进行报告。

任务 3 评价软件设施。SQA 应当对各种现有的和被计划的设施进行评估。对充分性的评价主要通过评价其是否为软件开发和软件支持提供了必要的仪器装备和空间。表 13-2 提供了支持项目开发的设施评估结果进行报告的样本格式。同时应当对执行这个任务的度量进行报告。

2. 软件过程审计

对规定的组织和项目的软件过程一致性审计是软件质量的关键部分。进行软件过程审计可以确保由组织和软件开发计划(SDP)定义的软件过程在项目进展的同时被遵守。这些软件过程审计通常采用的是对个别项目活动进行选择性的审核的形式,进而判断项目是否执行了每一项活动。

软件过程改进要求对项目使用的过程进行周期性地评估,以确定它们是适合的和有效的。在这些评估的基础上,SQA 应当标识出各种必要的和有益的过程改进,并标识出对 SDP、SCMP、SQAP,或其他控制项目执行的计划的改进建议,如果这些改进得到批准,则在项目上实现这些改进。下面是 SQA 在执行软件过程审计时的指导方针。

表 13-1 软件工具评估表格

<p style="text-align: center;">软件工具评估</p> <p style="text-align: center;">SQA: _____</p> <p style="text-align: center;">评估日期: _____</p>
被评估的软件工具:
评估方法和准则:
评估的结果:
建议的更改活动:
已采取的更改活动:

表 13-2 项目设施评估

<p style="text-align: center;">项目设施评估</p> <p style="text-align: center;">SQA: _____</p> <p style="text-align: center;">评估日期: _____</p>
被评估的项目设施(设备, 用户/测试/库空间):
评估方法和准则:
评估的结果:
建议的更改活动:
已采取的更改活动:

- 确定审计范围的方法：SQA 在进行审计的过程中，应当遵守规定的规程，以确保正确的审计步骤得到标识并提供最大的好处。SQA 审计规程应当对所有项目工作人员适用。这可以把 SQA 执行特定软件过程审计的范围和目的传达给所有人，例如 SQA 会审计项目的代码、单元测试用例、单元测试规程的同行评审过程。
- 过程符合性：对那些过程正在被遵守的区域，SQA 应当对这个过程活动满足活动和项目目标的有效性做一个对项目组有利的报告，如果需要，同时报告需要采取的建议活动（如果过程没有足够的健壮性，则对过程进行更改）。
- 过程不符合性：对过程未被遵守的区域，SQA 应当对不符合项、不符合项对项目的影响，以及项目组和项目管理部门应当采取的建议行动进行报告。

该项任务可以进一步被分解成下列更细的任务。

任务1 评估软件产品评审过程。软件产品评审过程中的 SQA 检查准备评审的软件产品是否得到评审，评审的结果是否得到了报告，被报告的结果或问题是否按照项目的 SDP 和规程得到了解决。软件产品可以包括信息的描述，而不是传统的硬拷贝文档。SQA 应当采用软件过程审计报告的形式报告对软件产品评审过程的评价结果，并报告执行该任务的度量。

任务2 评估项目计划与监督过程。项目计划与监控要求项目管理者形成以下计划文档。

- 软件开发计划：用于执行软件开发活动的计划，这些活动是软件开发标准所必需的。
- CSCI 测试计划：进行 CSCI 资格测试的计划。该计划写在软件测试计划文档中。
- 系统测试计划：进行系统资格测试的计划。对于软件系统，系统测试计划包含在软件测试计划中。
- 软件安装计划：在用户现场进行软件安装和培训的计划。
- 软件移交计划：一个标识各种支持机构需要的软件开发资源的计划。编制这个计划，既要标识为支持软件代理所必需的软件开发资源，又要描述移交可提交软件项时应该遵循的方法。

SQA 应确保项目执行了在上述项目计划中规定的相关活动。对这些计划的任何更改应提交更改申请，并得到项目管理者批准。编制计划文档时，SQA 应当标识出标准和指南，或者一个适当的数据项说明（Data Item Description, DID），并帮助对标准、指南或数据项说明进行裁减，以满足项目的要求。

任务3 评估系统需求分析过程。需求分析在顾客和软件项目组之间建立对顾客需求的共同理解。应该与顾客就软件项目的需求建立协议，并维护该协议。这个协议就是所说的将系统需求分配到软件和硬件的协议。系统需求被写在各种文档中，例如操作概念说明（Operational Concept Description, OCD），系统/子系统规格说明书（System/Subsystem Specification, SSS），以及接口需求规格说明书（Interface Requirements Specification, IRS）。在这个过程中，SQA 应当完成以下工作。

- 保证恰当的参与者被包含到需求的定义和分配过程中以识别所有用户需求。
- 确保需求得到了评审以确定它们是否切实可行，描述是否清晰，前后是否一致。
- 保证对已分配的需求、工作产品和活动的更改得到了标识、评审，并被跟踪到

结束。

- 保证涉及需求定义和需求分配过程的人员接受了适用于他们负责领域的、必要的规程和标准的培训，以便他们能够正确地完成工作。
- 保证来自于被分配需求的承诺经过与受影响的组协商并获得同意。
- 验证那些承诺是文档化、沟通过、评审过、并是公认的。
- 对被标识为有潜在问题的已分配的需求，应确保负责分析系统需求并形成文档的组重新复核了这些需求，并且进行了必要的修订。
- 验证规定的需求定义过程、需求文档编写过程和需求分配过程被遵循并形成文档。
- 保证配置管理过程用于控制和管理需求基线。
- 验证需求被文档化，并得到管理、控制和跟踪(更适合借助于矩阵法)。
- 验证同意的需求在软件开发计划(SDP)中可以查询到。

任务4 评估系统设计过程。系统级设计决定对系统行为的设计、对影响系统组件的选择和设计的其他方面。系统结构设计将系统组织成许多子系统，再将子系统组织成硬件配置项(Hardware Configuration Items, HWCI)、计算机软件配置项(Computer Software Configuration Items, CSCIs)和人工操作，或其他适当配置项的变体。系统设计被写成文档，例如系统/子系统设计描述(System/Subsystem Design Description, SSDD)和接口设计描述(Interface Design Description, IDD)。在系统设计过程中，SQA应当完成以下工作：

- 确保已准备了生命周期文档和需求跟踪矩阵，并保证它们的及时性和一致性。
- 保证与生命周期相关的文档在经过批准的需求变更的基础上进行了更新。
- 确保设计走读(同行评审)评价设计与需求的符合性，标识设计中的缺陷，并且抉择得到了评价和报告。
- 有选择地参与设计走读，并保证所有的走读得到了贯彻。
- 标识缺陷，验证以前标识缺陷的解决状态，并确保变更控制的完整性。
- 有选择地评审和审计系统设计文档的内容。
- 标识出与标准缺乏一致的项，并确定纠正行动。
- 判断需求及其相应的设计和工具是否与标准一致。对不符合项，在进一步软件开发之前是否需要被放弃。
- 评审演示原型与需求和标准的符合性。
- 确保演示与标准和规程相一致。
- 评审设计里程碑的状态。

任务5 评估软件需求分析过程。软件需求分析定义和记录了每一个CSCI满足的软件规格，用于保证每个需求被满足所使用的方法，以及CSCI需求和系统需求之间的可追踪性。软件需求被文档化到软件需求规格说明书(SRS)和接口需求规格说明书(IRS)中。在软件需求分析过程中，SQA应当完成以下工作：

- 确保软件需求定义和分析过程，以及相关的需求评审被根据项目制定的和在项目SDP中描述的标准和规程管理和执行。
- 确保从软件需求分析中确定下来的行动项能按照这些标准和规程得到解决。

任务6 评估软件设计过程。初步设计活动确定了要建立软件的整体结构。根据在前

一阶段标识的需求,软件被分割成许多模块,并且每个模块的功能和那些模块间的关系被定义。详细设计用于定义和完成软件逻辑的详细的设计,以满足已分配的需求。详细设计应该详细到可以由其他工程师,而不是由原始设计者完成计算机程序编码的程度。软件设计应当以软件设计描述(Software Design Description, SDD)、数据库设计描述(Database Design Description, DBDD)和接口设计描述(Interface Design Description, IDD)等的形式编制成文档。在软件设计阶段, SQA 应当完成以下工作:

- 确保软件设计过程以及相关的设计评审按照项目制定的、在项目 SDP 中描述的标准和规程进行。
- 确保来源于软件设计评审的行动项按照这些标准和规程加以解决。
- 为了确定作为评估软件单元开发过程管理工具的方法的有效性,对用于跟踪和文档化软件单元开发的方法进行评估。应用于这个评估的实例标准应包括进度表信息、审计结果以及内部评审和批准被评审对象的指示。
- 确保那些用于跟踪和文档化软件单元开发的方法,例如软件开发文件(SDF)或单元开发文件夹(UDF),得到实施并保证文档的及时性。

任务7 评估软件实现与单元测试过程。软件实现或编码是软件开发周期中软件设计的实现阶段。这个过程包括软件单元测试。在编码和单元测试过程中, SQA 应当完成以下工作:

- 确保软件编码过程、相关的代码评审和软件单元测试按照标准和规程开展进行。
- 确保来自于代码评审的行动项按照项目规定的和项目 SDP 中描述的标准和规程被解决。
- 确保用于跟踪和文档化软件单元开发的 SDF/UDF 得到实施并保持了文档的及时性。

任务8 评估单元集成与测试、CSCI 资格测试、CSCI/HWCI 集成与测试,以及系统资格测试。软件集成和测试活动在开发环境中将每个单独开发的组件结合起来以保证组件能够共同工作以完成软件和系统功能。为了将硬件和软件开发结合起来,集成要求硬件和软件紧密的同步以满足设定的集成和测试里程碑。在软件开发生命周期的集成和测试阶段,测试的焦点从单个组件的正确性转移到了组件间接口的正确操作、通过系统的信息流,以及系统需求的满足程度上。在集成和测试阶段, SQA 应完成以下工作:

- 保证软件测试活动被标识,测试环境已经被定义,并且用于测试的指导书已经被设计。SQA 应验证软件集成过程,软件集成测试活动和软件性能测试活动是按照 SDP、软件设计、软件测试计划,以及规定的软件标准和规程执行的。
- 确保任何向执行软件集成测试或软件性能测试的人员的代码转交都按照规定的软件标准和规程完成的。
- 确保尽可能多的必要的软件集成测试和所有软件性能测试以证明已批准的测试规程被得到遵循,测试结果的正确记录被保存,测试期间发现的所有差异被完整地报告,测试结果得到了分析,并且相关的测试报告被完成。
- 确保纠正行动过程能有效地标识、分析、更正在软件集成和性能测试期间发现的所有差异;在必要时重新执行软件单元测试和软件集成测试,以确认对代码做了正确的更改;确保软件单元的设计、代码和测试以软件集成测试、软件性能测试

和纠正行动过程的结果为基础进行了修正。

- 确保软件性能测试确定了软件的性能参数。
- 确保测试和报告结果责任已经分配给特定的组织成员。
- 确保已制定了测试监控规程。
- 评审软件测试计划和软件测试规程与需求和标准的符合性。
- 确保软件被测试。
- 监控测试活动，提供测试证据，证明测试结果的合格性。
- 确保已经制定了证明或校准在测试期间使用的所有支持软件或硬件的需求。

任务9 评估结束项提交过程。这个活动可应用于那些提供一次性交付产品的项目，也可以用于在特定时间周期或时间段内提供必要的交付产品的项目。SQA 应当评估那些用于最终交付项准备的活动，以确保用于结束项产品的功能和物理审计的程序或项目需求都得到了满足。在某些情况下，如果项目没有满足程序、项目需求或标准的要求，则应当允许 SQA 禁止某些项的交付，例如文档、代码或系统。

任务10 评估纠正活动过程。纠正行动过程可描述为以下步骤：

- 为确保尽早发现实际或潜在的问题，在软件开发期间就对问题进行标识和纠正；
- 将问题报告给适当的权威人士；
- 分析问题，建议纠正方法；
- 及时完成纠正活动；
- 记录并追踪每一个问题的状态。这里的问题包括：文档错误、软件错误，以及与标准和规程的不符合性。

在这个任务中，SQA 应当完成以下工作：

- 为了评价纠正行动过程的有效性，依据 SCMP，周期性地对纠正行动过程及其结果进行评审。
- 周期性地分析所有报告的问题，识别一般的问题区域。这些分析应包括原因研究、影响幅度、发生频度和预防措施。

任务11 介质认证。SQA 应当保证交付给软件获取机构的包含源代码的介质和包含目标代码的介质彼此一致。SQA 也应当保证在这个介质中存储的软件版本与已完成软件性能测试的软件版本相匹配，或者这个介质正确存储了一个经授权的代码升级，并且是可以使用的。SQA 报告与纠正行动记录、软件测试报告，以及软件产品评估记录一起，可以构成软件介质认证必需的依据。

任务12 非交付软件认证。在可交付软件开发中，项目可以使用非交付软件，只要可交付软件在交付给需要方后其操作和支持不依赖于非交付软件，或者已预备需要方有或可以获得同样的软件。SQA 应当确认非交付软件的使用是满足上面的标准的，也就是，可交付软件不依赖于非交付软件就可以执行，或者确认需要方可以获得相同的软件。SQA 应当确认所有用于项目的非交付软件能执行设想的功能。SQA 报告应与纠正行动记录、软件测试报告，以及软件产品评估记录一起，能够构成证实软件能尽到设想的职责必须的证据。

任务13 评估存储和处理过程。SQA 应当证明存在一个已经确立的与介质的存储和处理有关的计划、方法或过程集。SQA 应当评价软件产品和文档的存储以确保用于纸件

产品或介质的存储区域能够避免不利环境的影响,例如高湿度、磁力、灰尘的影响。

任务 14 评估子合同控制。SQA 应当负责确保来自于子合同的所有软件产品的质量符合合同要求并跟踪子合同的配置管理计划和规程。

任务 15 评估偏差和放弃过程。如果必要, SQA 应当帮助程序或项目管理者管理偏差和放弃的请求, 并证明这些偏差或放弃申请按照项目的 SCMP 得到处理和审批机构的批准。

任务 16 评估配置管理过程。配置管理是纪律, 它应用技术的和行政的管理和监督到: (1) 标识并文档化配置项(CI)的功能和物理特征。(2) 对配置项及其相关文档的更改进行控制。(3) 记录和报告需要的信息来有效管理计算机软件配置项(CSCI), 包括建议的更改项的状态和已批准的更改项的实施状态。(4) 审计配置项以验证其与规格、接口控制文档以及其他需求的一致性。在该任务中, SQA 应当评估以下内容:

- 确保已对文档、代码和计算机数据的配置标识制定了标题、命名、描述变更状态的标准。
- 确保开发基线变更(包括文档、代码、和计算机数据)的基线管理按照规定的规程进行了标识、评审、实现和合并。
- 确保基线文档和软件变更的配置控制按照 SCMP 的特定配置管理要求进行管理。
- 确保在规定的时间内, 按照规定的规程准备了配置状态总结报告, 并报告了与软件产品和文档的配置管理密切相关的配置项的状态。
- 确保指定参与配置审计的人员遵守 SCMP。
- 对文档控制, 应确保被项目组成员使用的是惟一批准的, 并且是最新的文档, 且保证文档发布过程使文档接收者收到正确的文档。
- 确保程序支持库是惟一的、存储所有软件的基线版本的位置。确保所有软件标识包括了软件名称和惟一的版本标识号。评估也应当确保对软件产品的存取控制经过了适当地操作运用, 确保对主要文件不能发生非授权的更改。

任务 17 评估软件开发库控制过程。软件开发库函数是软件配置管理主要的控制点。一个软件开发库包括所有形成计算机软件配置项(CSCI)的代码单元, 同时还包括仔细标识了的清单、补丁、勘误表、CSCI、系统磁带和磁盘包, 以及操作和建立软件系统的工作控制流。软件开发库还应包括以磁带或磁盘形式保存的、以前版本的可操作运行的软件系统。在这个过程中, SQA 应当完成以下工作:

- 确保软件开发库和管理软件开发库操作的规程已经建立。
- 确保文档和计算机程序资料经过了核准, 并放在软件开发库控制之下。
- 确保为已经批准的配置管理文档和软件版本建立了正式的发布规程。
- 确保软件开发库控制禁止对受控制软件的非授权更改, 并保证所有已批准的更改合并到了软件开发库。

任务 18 评估非开发软件过程。非开发的软件(NDS)是指由签约人、政府或第三方提供的软件。SQA 应当验证非开发的软件完成它被设想的功能。

任务 19 执行配置审计。SQA 可以被要求按照项目的软件配置管理计划执行或帮助进行一个正规的配置审计。配置审计是计算机软件配置项(CSCI)的正规检查。存在两种类型的配置审计: 功能配置审计(FCA)和物理配置审计(PCA)(参考 MIL-STD-973)。

任务 20 验证需求管理 KPA 的实现。需求管理的目的是在顾客和软件项目组之间建立一个对软件项目需求的共同理解。SQA 组应对用于管理已分配需求的活动和工作产品进行评审和/或审计,并报告结果。

任务 21 验证软件项目计划 KPA 的实现。软件项目计划的目的是为实施软件和管理软件项目建立合理的计划。SQA 组应对用于软件项目计划的活动和工作产品进行评审和/或审计,并报告结果。

任务 22 验证软件项目跟踪与监督 KPA 的实现。软件项目计划跟踪与监督的目的是为实际的软件开发过程建立恰当的可视性,以便当软件项目的执行明显偏离了软件计划时,管理者可以采取有效的行动。SQA 组应对用于软件项目跟踪的活动和工作产品进行评审和/或审计,并报告结果。

任务 23 验证软件子合同管理 KPA 的实现。软件的子合同管理的目的是选择合格的软件承包商,并有效地管理它们。SQA 组应对用于软件子合同的活动和工作产品进行评审和/或审计,并报告结果。

任务 24 验证软件配置管理 KPA 的实现。软件配置管理目的是在整个项目的软件生命周期内,建立并维护软件项目产品的一致性。SQA 组应对用于软件配置管理的活动和工作产品进行评审和/或审计,并报告结果。

任务 25 验证组织过程定义 KPA 的实现。组织过程定义的目的是开发、维护一套可以使用的软件过程资产,以改进整个项目的过程性能,并提供一个基础用于组织的长期的获益。SQA 组应对用于开发和维护组织标准软件过程和相关过程资产的活动和工作产品进行评审和/或审计,并报告结果。

任务 26 验证集成软件管理 KPA 的实现。集成软件管理的目的是集成软件和管理活动成一个连贯的、确定的软件过程。这个过程是从组织的标准的软件过程和相关软件过程资产中经裁剪后形成的。在组织过程定义中对组织的标准的软件过程和相关过程资产做了说明。SQA 组应对用于管理软件项目的活动和工作产品进行评审和/或审计,并报告结果。

任务 27 验证软件产品工程 KPA 的实现。软件产品工程的目的是不断地执行一个良好定义的工程过程以便越来越有效地生产出正确的、一致的软件产品。SQA 组应对用于软件产品工程的活动和工作产品进行评审和/或审计,并报告结果。

任务 28 验证组间协调 KPA 的实现。组间协调的目的是建立一种让软件工程组积极参与其他工程组活动的方法,这样项目可以更好地满足顾客的需要。SQA 组应对用于组间协调的活动和工作产品进行评审和/或审计,并报告结果。

任务 29 验证同行评审 KPA 的实现。同行评审的目的是尽可能早地移去软件工作产品中的缺陷。SQA 组应对用于同行评审的活动和工作产品进行评审和/或审计,并报告结果。

3. 技术和管理评审

由项目组进行的技术评审主要用于在项目生存期间生产的关键软件产品;举行管理评审则是为了检查和讨论项目的状态、软件产品和/或项目存在的问题。何时进行技术评审和管理评审依赖于主要的“里程碑”、选定的程序策略和开发方法,以及高级管理者和项

目经理的决定。SQA 功能提供的输入确保了被标识的评审对项目来说是充分的。另外, SQA 应参与评审, 并报告评审的结果和度量。

该项任务可以进一步被分解成下列更细的任务。

任务 1 参与技术评审。工程质量进入到软件产品中去的一个主要组成部分是对软件产品进行技术评审, 软件产品既包括可提交的软件, 也包括非提交的软件。技术评审的参与者应当包括对被评审的软件产品的技术领域专家。技术评审应当关注的是过程中的和最终的软件产品, 而不应当是为了评审专门编写的资料。SQA 功能不但应确保项目组有效地完成了相关的技术评审, 同时应当按照已批准的采样技术, 有选择地参加技术评审。技术评审的类型和范围很大程度上依赖于软件产品的规模、范围、风险性和关键性。技术评审有助于通过一边工作一边培训的方式来增加有效性, 并且技术评审允许使用统一的方法评审工作产品及其状态。这种类型评审的目标是:

- 评审正在逐步形成的软件产品。对建议的技术解决方案进行审核和论证。对该技术效果提供洞察力, 并获得反馈。使技术问题表面化并加以解决。
- 评审项目状态。将与技术、费用和进度问题相关的近期和长期的风险表面化。
- 对已标识风险的减缓策略, 需要在所有评审参与者之间达成一致。
- 标识在管理评审点提出的风险和问题。
- 确保在软件获取方与开发方技术人员之间进行相互交流。

评审通常发生在每一开发周期阶段的结束, 并集中在是否该软件产品应当提交到开发的下一个阶段。MIL-STD-1521B 定义了以下技术评审内容。

系统需求评审(System Requirements Review, SRR):目标是确保在定义系统技术需求和实现其他工程管理活动中的进展。这类评审的数量是由获取系统需求的活动决定的。系统需求评审一般在系统功能需求已经被建立的时候执行。

系统设计评审(System Design Review, SDR):目标是评价与已分配的技术需求相关的优化性、相关性、完整性和风险。还包括对系统工程过程和下阶段工作的工程计划的一个总结性评审。这里的系统工程过程用于产生已分配的技术需求。这个评审一般在系统定义过程已经完成系统特性的定义并产生相关配置项时进行。一个成功的系统设计评审需要在所有参与者之间就以下方面达成一致意见:(1)最新的或完整的系统或子系统规格说明书;(2)完整的配置项开发和关键项的规格;(3)其他系统定义工作、成果和计划。

软件规格说明书评审(Software Specification Review, SSR):目标是对最终定稿的 CSCI 需求和操作概念进行评审。SSR 一般在 CSCI 已经被充分定义时执行。一个成功的 SSR 为进入软件初步设计建立了一个包括软件需求规格说明书、接口规格说明书和操作概念文档的基础。

初步设计评审(Preliminary Design Review, PDR):目标是对每个 CI 或 CI 的集合进行评估。(1)评价在已选择的设计策略下的进展、技术充分性和风险。(2)确定它们与 CI 开发规格的性能需求及工程专业需求的兼容性。(3)确定在 CI 与其他设备、工具、计算机程序和人员之间物理和功能接口的存在方式和兼容性要求。对于 CSCI, 这个评审关注于:①评价选定的顶层设计和测试方法的进展、一致性和技术充分性。②评价软件需求和初步设计之间的一致性。③评价业务和支持文档的先前版本。

关键设计评审(Critical Design Review, CDR):目标是在详细设计全部完成时对每个

CI 进行评估。该评审用于：(1)确定评审中的 CI 的详细设计满足 CI 开发规格的性能需求和工程专业需求。(2)确定详细设计在 CI 与其他设备、工具、计算机程序和人员之间的兼容性。(3)评价 CI 的风险区域。(4)评价系统硬件的可生产性分析结果。(5)评审原始硬件产品规格。对于 CSCI，这个评审关注于确定设计解决方案的详细设计、性能和特性的可接受性，并且还关注业务和支持文档的充分性。

测试准备度评审(Test Readiness Review, TRR)：目标是对每个 CSCI 确定其软件测试过程是否完整，并且保证承包商为正式的 CSCI 测试进行了准备。TRR 评价软件测试过程是否遵从软件测试计划，并且足以完成测试目标。在 TRR 中，合同代理方还要评审非正式的软件测试结果和任何业务及支持文档的更新情况。一个成功的 TRR 为进入正式的 CSCI 测试提供了一个包括软件测试过程和非正式测试结果的基础。

功能配置审计(Functional Configuration Audit, FCA)：目标是为了验证一个配置项的开发已经被圆满地完成，并且拥有了特定的功能及性能属性。此外还要评审完整的业务和支持文档。

物理配置审计(Physical Configuration Audit, PCA)：目标是验证一个指定的配置项是否遵循了定义该配置项的技术文档的要求。

正式资格评审(Formal Qualification Review)：目标是验证构成系统的一组配置项的测试、检视和分析过程是否满足合同要求的业务需求。对于单个配置项，该评审不会应用到在 FCA 中验证过的硬件或软件需求上。

产品准备度评审(Production Readiness Review, PRR)：目标是确定特定的软件活动的完成状态。这些软件活动必须在产品执行下一步决定之前被很好地完成。在工程和制造开发阶段，PRR 主要以一种递增的方式进行，一般最初的两个评审和最后的一个评审主要评价在产品执行下一步决定时的风险。

对每一种软件产品选择进行什么类型的技术评审，评审哪些项目，以及每个评审的参加人员应当在软件开发过程、项目 SDP 和项目 SQAP 中加以描述。应当通过软件过程审计，对评审过程与规定的过程的符合性进行验证。

如果正规的软件过程、SDP、SQAP，或者项目职员和项目管理者认为是适当的，那么技术评审可以被执行在软件产品的任何一个部分上。典型的被评审项目包括需求规格、接口规格、功能描述、设计规格、源代码、测试计划、测试数据、测试用例、测试规程、用户文档，以及维护手册。下面是用于技术评审的准则：

- 评审项目完整性：确定评审项是否完全满足其被设想的目标。
- 问题标识：尽可能早地标识问题，以便在问题被混合到随后的项目阶段之前，能够修正它们。
- 与标准的符合性：确保评审项与已确立的或要求的标准相符合，或者找出没有满足标准的不符合项。
- 风险标识：标识在项目内潜在的风险区域，以便在项目进行的过程中，风险可以得到管理、减轻。
- 可跟踪性：确保项目可以跟踪到项目的以前的、合适的阶段。通过跟踪矩阵记录项目的可跟踪性，对开发者和管理者都是非常重要的。他们利用跟踪矩阵可以核实软件满足了他们的需求，且提供了随后的项目活动的输入。

任务2 参与管理评审。SQA 对软件项目状态、过程、问题，以及风险的周期性管理评审为软件项目活动提供了独立的评价。这类参与要求 SQA 准备向管理者提供下面信息：

- 符合性：标识项目与组织已确定的、及项目自己确定的软件过程的符合性级别。
- 问题区域：在对技术评审结果进行分析的基础上，标识出潜在的，或实际的项目问题区域。
- 风险：在参与和评估项目过程和问题区域的基础上，标识出软件的风险。

因为 SQA 功能对项目成功是必不可少的，所以 SQA 应当能够自由地通报 SQA 活动的结果给高级管理者、软件工程过程组 (SEPG)、项目管理者 and 项目组。在管理评审期间，SQA 可以经常给项目经理提供额外的支持，以便强调高级管理者关注的区域。

报告符合性、问题区域和风险的方法应当以文档化的报告或备忘录的形式传递。对符合性、问题区域和风险应当进行跟踪，并应跟踪到结束。

4. SQA 报告

一个重要的 SQA 活动是报告每一个软件产品评估或软件过程审计的结果。SQA 应当将已完成的 SQA 活动结果文档化，并迅速地启动更改活动或提供 SQA 认证。将 SQA 报告结果提供给适当的程序人员/项目人员。

该项任务可以进一步被分解成下列更细的任务。

任务1 准备软件产品评价报告。对已进行过的任何软件产品评估，都要求有评估记录。对软件产品评估记录的要求是，它们必须被准备，并且对受影响的程序人员和项目人员是可用的。作为最低限度，评估记录应当包括：

- 被评估的产品；
- 用于评估的方法和准则；
- 评估的结果；
- 建议的更改活动；
- 采取的实际更正活动。

项目的 SDP 应当标识所有被开发的软件产品，以及可能引用的数据项描述或者项目应当遵守的、专为项目裁减的数据项描述。软件产品评估标识了要求经受软件产品评估的软件产品，标识了每一个评估应当使用的准则，也包括了一套默认的评估准则定义。

软件评估记录的格式可能与进行软件评估采用的具体过程有关。假如项目评估软件产品使用的是作为软件更改活动过程输入的软件问题报告的格式，则应当在软件配置管理计划中，对该更改活动过程，以及软件问题报告的格式加以说明。

软件产品评估的 SQA 报告应当在 SQAP 中编制成文档，在 SQAP 中阐述，SQA 应当评估什么软件产品，如何进行评估（评估的方法和准则），用什么形式报告评估的结果，评估的结果则应包括推荐采纳的软件更改活动和实际被采纳的更改活动，被评估的软件产品在标题为“软件产品、工具和设备的 SQA”段落中，做了更进一步的论述。

任务2 准备软件过程审计报告。每一次软件过程审计都要生成一个软件过程审计报告，它既展示了正在得到正确地遵守、并有效地工作的软件过程区域；也展示了虽然得到遵守、但是没有能有效地工作的软件过程区域；以及根本没有得到遵守的软件过程区域。SQA 的建议可能会导致对软件过程进行调整，或者促使项目按照已确立的计划和过程，执

行已打算准备实施的软件活动。表 13-3 提供了一个过程审计报告的样本格式。软件过程

表 13-3 过程审计报告

软件工具评估					跟踪标识:
审计领导:					
报告日期:					
审计小组:					
项目名称:					
审计日期:					
被审计的过程/规程:					
审计使用的检查表:(附件)					
审计发现:(请在其中一项前打钩) <input type="checkbox"/> 过程/规程可以接受 <input type="checkbox"/> 过程/规程可以有条件地接受 (需要满意地完成的活动项清单如下) 条件说明: <input type="checkbox"/> 过程/规程不可以接受 (需要满意地完成的活动项清单如下) 条件说明:					
活动项(AI):					
AI 编号	标题:	分配给:	预计日期:	完成日期:	
更改活动:					
处理意见: 批准 取消 延期					
项目经理:					日期:
活动项结束:					
SQA 签名:					日期:
(该文件以 SQL 评估记录的形式完成)					

审计报告被直接送给:

- 高级管理者——软件过程审计的结果与其他的项目状态信息一起,可用做高级管理者的管理向导,使其注意在组织层次上标识并减轻项目的风险。
- 软件工程过程组(SEPG)——软件工程过程组(SEPG)利用过程审计结果和其他项目的审计结果一起,标识出软件过程的弱点,并展开或增强特定区域的过程改进。这个报告数据也可变成过程数据库的一部分,可用于将来项目的分析和使用。
- 项目经理——项目经理利用这个报告,可以洞察到项目过程是否与开发过程相符合,是否有效地满足了项目的目标。在必要和适当的软件过程点,项目经理可以启动相应的实施活动,或者利用批准的规程对已确定软件过程的启动变更活动过程。

5. SQA 度量

表 13-4 提供被做并被用于确定 SQA 活动进度和费用状态的度量。

表 13-4 SQA 度量

测量的内容	应收集的度量
SQA 里程碑数据(计划的)	报告计划
SQA 里程碑数据(完成的)	报告实际与计划
列入计划的 SQA 工作(计划的)	报告计划
完成的 SQA 工作(实际的)	报告实际与计划
SQA 花费工作量(计划的)	报告小时数
SQA 花费工作量(实际的)	报告小时数
SQA 花费的资金(计划的)	报告每人年的人民币(元)
SQA 花费的资金(实际的)	报告每人年的人民币(元)

该项任务可以进一步被分解成下列更细的任务。

任务 1 收集、报告软件产品评估度量。这个活动应记录 SQA 进行软件产品评估相关的工作量。应收集、记录的度量数据包括每个人评估软件产品的工作量(小时数),还包括软件产品评估的调查结果。对该度量的记录应当可以展示在执行一个特定的 SQA 活动时, SQA 的工作时间分配。表 13-5 提供了一个 SQA 评估一个需求规格书并报告评估结果的样本。

表 13-5 需求评估样例

软件产品	页数	用于评估的小时数	用于报告的小时数
软件需求规格(SRS)	20	3	1

任务 2 收集、报告软件产品质量度量。一个可以收集的质量度量是,在软件产品评估中报告的错误类型和数量。为确保软件质量, SQA 人员应对这个度量进行分析。例如,如果 SQA 人员发现大量的需求或设计错误,则应当建议软件开发小组采取根源分析活动。

在进行原因分析时，应当考虑：为什么产品评审者当初没有能够捕捉到需求或设计错误。另一个调查区域就是看检查软件过程，搞明白软件过程是否足够的健壮，对包括评估产品在内的评审人员是否是合适的，是否有太多的含糊不清等。

任务3 收集、报告软件产品过程审计度量。这个软件活动要求对一个项目执行软件过程的 SQA 审计所花费的时间进行管理并报告。表 13-6 是一个 SQA 评估项目的更改活动过程的过程审计度量样本。

表 13-6 过程审计度量表格样例

被审计的软件开发过程	用于审计准备的时间(小时)	用于评价的时间(小时)	用于报告的时间(小时)
更改活动过程	2		1

13.2.3 产生/维护 SQA 计划

这个 SQA 活动详细描述了生成和维护一个项目的软件质量保证计划 SQAP 的过程。SQAP 的目的是描述开发人员计划如何确保软件产品的质量满足质量方针政策确定的那些标准，SQAP 也详细描述了被采用的质量规程和技术。这个 SQA 计划的目标就是要能够说明开发者将如何确保软件系统的需求得到满足。首要的质量保证任务就是要下决心，在项目执行期间，贯彻这些质量保证活动，这就意味着，在项目的每一个开发阶段，SQAP 必须明确地定义，应当如何执行质量保证活动(例如设计评审和系统测试)。

作为一个最低的极限，SQAP 应当描述：

- 质量目标，不论什么时候，如果可能则明确可测量的条款；
- 测试类型标识，对产品规格、计划和测试规格，以及被采用的方法和工具，执行验证和确认活动；
- 每个开发阶段，定义的进入准则和退出准则；
- 应当完成的、详细的测试验证和确认活动计划，包括进度、资源和批准权；
- 质量活动，例如检视、评审和测试，配置管理和更改控制，测量和报告，缺陷控制和更正行动的特定的责任；
- SQA 组的职责和权利；
- SQA 组的资源要求(包括职员、工具和设施)；
- 项目 SQA 组活动的时间表和资金；
- 在建立项目的软件开发计划、标准和规程时，SQA 应参与的活动；
- SQA 组应当履行的评估；
- SQA 组应当进行的审计和审核；
- 项目标准和规程用做 SQA 组审核和审计的基础；
- 文档化、并跟踪不符合问题到结束的规程；
- 要求 SQA 组编写的文档；
- 向软件工程组和其他软件相关组，提供有关 SQA 活动的反馈的方法和频度。

生成和维护一个项目的 SQAP 过程可以按照下面的步骤进行。

1. 进入准则

“进入准则”定义为启动一个规程时，必须到位的、必要的要素和条件。这个规程的进入准则是：

- SQA 方针政策；
- 项目需求；
- 项目软件开发计划(SDP)；
- 项目的软件产品，例如规格，标准、规程；
- 已标识的 SQA 活动；
- 已分配的 SQA 任务；
- 协商、实施项目 SQA 的职责得到明确的分工；
- SQA 人员、小组、机构有一个向高级管理者的报告渠道，它独立于所有与项目相关的其他组；
- 软件项目计划的适当资源和预算，已经被标识，并已分配；
- 与 SQA 活动相关的人员，接受了 SQA 目标、规程和方法的培训与技巧训练。

2. 控制

“控制”定义为约束或管理一个过程活动的的数据。“控制”管理输入到输出的转变。下面是生成和(或)维护 SQAP 的“控制”。

- 软件质量保证计划 SQAP 指南；
- 质量程序的需求；
- SQA 方针政策。

3. SQAP 评审

SQAP 必须得到项目的所有人员的审核，这就要求项目的每一个人员，对在 SQAP 中描绘的角色、职责、资源和活动，提交评论与反馈。应当对 SQAP 进行评审，评审可以遵循“正规检视过程”的方式，也可以遵循“非正规文档评审过程”的形式。

4. SQAP 批准

SQAP 的批准过程应当得到程序或项目管理部门，以及配置控制委员会的批准。

5. 配置控制

已批准的 SQAP 应该放在配置控制之下。为了反映当前项目的需求、过程和实践，SQAP 进行的修订必需对其审核和批准。

6. 测量

在开发 SQAP，包括对 SQAP 的所有升级过程中，应当记录工作人员的级别，劳动时间，以及开始和停止日期。

7. 退出准则

退出准则被定义为, 完成一个规程必须具备的要素和/或条件。这个过程活动的退出准则是 SQAP 已被批准, 并被置于配置管理之下。

8. 对产生/维护一个项目的 SQAP 规程进行过程改进

对项目组人员遵循这个产生、维护项目的 SQAP 过程的紧密程度进行审核, 然后将需要改进的过程区域提供给那些负责过程改进的人员。过程生成 SQAP 的效率可能需要对这个过程升级。

13.2.4 实施 SQA 计划

这个 SQA 活动的目的是按照项目已批准的 SQAP 履行 SQA 功能。

13.2.5 产生/维护 SQA 规程

这个活动的目的是要文档化和维护 SQA 执行的过程, 这个过程描述了 SQA 如何依据当前已批准的 SQAP 在项目中的执行。这个过程活动的目标就是要展示履行一个特定任务的“可重复性”, 所以, 执行任何 SQA 活动都应当编写一个文档化的规程。

SQA 经理负责起草或指导 SQA 规程的开发, 实施并维护 SQA 规程, 以反映本项目的 SQA 活动是按照项目 SQAP 中描述的方式进行的。

生成/维护 SQA 规程的输入包括已标识的任务(WBS)和 SQAP。这个活动的输出是已文档化的 SQA 规程。文档化的 SQA 规程既可以作为附录包括在 SQAP 中, 也作为一个单独的、从 SQAP 中独立出来的文档存在。软件工程室建议将这些规程从 SQAP 中独立出来, 因为过程文档不像 SQAP 那样, 它没有必要进行严格的更改管理。采用这个建议的前提是, 假定 SQAP 已经得到了项目配置控制委员会(CCB)的审核和批准, 并且正式放在了配置管理之下。

在定义一个规程应当包含什么内容时, 理解规程的目标是什么是非常重要的。一个规程的目标应当是能够提供某个人或某些人执行 SQA 任务的一步一步的命令指示。

13.2.6 标识 SQA 培训

SQA 人员需要标识出各种对执行 SQA 任务可能是必需的培训。项目的需求也会提出 SQA 人员需要培训什么内容, 使得 SQA 人员能在项目中恰当地履行 SQA 功能。

软件工程室能够提供的培训, 可以参看软件工程室的相关培训公告, 也可以直接与软件工程室联系相关培训。

13.2.7 标识/选择 SQA 工具

该部分活动主要用于描述选择支持软件质量, 或 SQA 功能的工具的步骤。SQA 工具选择规程可以描述如下:

- 定义项目的 SQA 需求。定义 SQA 需求的一个先决条件是项目的需求必须已经确定。在 SQAP 中定义的 SQA 需求, 其基础是在项目的软件开发计划(SDP)中确定的项目需求。
- 开发一个选择 SQA 工具的活动和里程碑的计划。
- 安排一个希望选择供应商的预定日期。
- 安排一个产品 DEMO 的试验日期。
 - ✓ 安排工具评估和选择最好的供应商的日期。
 - ✓ 安排购买工具的日期。
 - ✓ 安排工具建立安装的日期。
 - ✓ 安排正规培训的日期。

标识预定的你希望工具到位投入使用的日期。

- 建立工作组, 如果可能的话, 帮助项目在项目需求和 SQA 需求的基础上标识并选择工具。
- 定义并文档化 SQA 工具需求。需求应当包括运行软件必须的计算机平台, 该平台的操作系统 SQA 工具与软件工程环境的接口, 被开发软件的类型, 例如 Ada, C 等, 目标计算机等。应当考虑包括项目能够承受的工具费用支出有多少, 支持工具的资源, 决定购买还是公司内部开发工具, 做费用收益分析等。
- 阅读产品文献。了解需要什么样的工具。一个信息来源是软件技术中心有关工具的报告, 软件技术支持中心的报告可以在 SEPO 的库中获得。另一个信息来源是在 WWW 上冲浪。
- 缩小工具的选择范围到一个可以操作的程度(3~5 个可能的供应商), 这些工具更好地满足了 SQA 工具的需要。
- 向供应商申请产品 DEMO。向供应商提供一个 SQA 工具需求的拷贝。请供应商就工具满足在 SQA 工具需求文档中明确指定的需求的能力进行演讲答辩。
- 申请获得 SQA 工具软件的评估备份。并在评估期间申请获得支持。
- 在对着 SQA 工具需求文档进行工具评估的基础上, 对工具进行估价打分, 打分范围可以从 1~5 分, 5 分是最高分。最后做出购买工具的决定。

13.2.8 改进项目 SQA 过程

软件过程改进的 SQA 活动要求:

- 理解项目的 SQA 过程。
- 确定哪里发生了无效率的事例或缺陷(缺陷的根源)。
- 对项目过程提供更改建议, 以提高效率或减少缺陷。

- 提供更改建议，以排除缺陷的根源。
- 为项目组提供培训课程建议。

这个活动的目的是为 SQA 评审现存的项目和 SQA 过程，报告过程改进的有效性区域，并标识需要明确的过程。为了改进项目的 SQA 过程，SQA 需要审核、评价项目过程和 SQA 过程。这将确保项目过程和项目的 SQA 过程彼此是一致的并且是兼容的。改进项目 SQA 过程的方法是由 SQA 进行内部 SQA 审核(评审)。内部 SQA 审核(评审)需要遵守公司的 SQA 方针政策，遵守项目的 SQAP 和 SDP，并遵守文档化的 SQA 过程和规程。报告软件过程审计调查结果的方法是过程审计报告。过程审计报告应当用做过程改进开始行动。过程改进有可能导致 SQA 方针政策、过程、和/或规程的变更。

13.3 本章小结

软件质量保证是软件开发过程中的一个关键性活动，目的是保证开发过程沿着既定的规范过程开展，发现过程中出现的不一致问题，并试图寻找一种过程改进的方法。软件质量保证给管理者提供了一个项目开发过程及正在构造产品的视角。这样，管理者就能够及时知道产品中可能存在的问题，而不是到最终结束时才发现产品偏离目标。当软件质量保证在项目组内部发现不符合性问题时，首先是试图解决这些问题，对于那些无法在软件项目内部解决的问题，软件质量保证组逐级递交该问题到管理者的恰当层次以求解决。

好的软件实践要求独立的软件质量保证小组，这种独立性为软件质量保证提供了关键的力量支持。也就是说，如果产品的质量受到危害时，软件质量保证小组有权直接报告这种可能性给项目的高级管理者。同时软件质量保证活动必须在软件质量保证计划的指导下开展活动。软件质量保证计划的目的是描述开发人员计划如何确保软件产品的质量满足质量方针政策确定的那些标准，同时该计划也详细描述了被采用的质量规程和技术。

第14章 需求测试

传统的软件动态测试已经发展到了全面的软件测试。这个测试不仅仅针对代码运行的测试，还包括文档和需求的测试。尽管需求测试还处在一个探索阶段，更多的还只能借助于测试者的经验和技能。但是，我们还是希望能够从中提取一些共性的东西，以供参考和研究。本章将介绍需求测试方面的一些概念和方法，包括需求本身以及需求分析使用到的一些技术方法。通过本章的学习，你将了解以下内容：

1. 什么是需求？
2. 需求的 FURPS + 模型内容是什么？
3. 一个好的需求具有哪些特点？
4. 本文提到哪几种需求测试方法，它们分别是如何进行的？

14.1 需求测试概述

软件测试 V 模型要求在需求阶段就开始制定系统的测试计划，开始考虑系统的测试方法。但仅有这些还是不够的，全面的质量管理要求在每个阶段都要进行验证和确认。因此，在需求阶段还需要对需求本身进行测试。这种测试是必要的，因为在许多失败的项目中，70 % ~ 85% 的返工正是由于需求方面的错误所导致的^[208]。并且，因为需求的缘故而导致大量的返工，造成进度延迟、缺陷发散，这是一件极其痛苦的事情。因此，要求在项目的源头(需求)就开始测试，这类测试更多的还只是静态手工方式的测试，当然也有一些自动化的工具。但这些工具会要求测试者按照某个固定格式进行需求的表述(例如形式化的方法)。因此，在适用性上会受到限制。在用静态手工方法进行的需求测试中，最常使用的手段是同行评审。关于同行评审的概念和过程将在第 16 章进行详细讲解。本章关注的是从哪些方面去测试需求、如何测试需求。

14.1.1 什么是需求

在讲解需求测试之前，首先需要对需求有一个清晰的了解。什么是需求呢？

IEEE 软件工程标准词汇表(1997 年)中定义的需求为：

- (1) 用户解决问题或达到目标所需的条件或能力(Capability)。
- (2) 系统或系统部件要满足合同、标准、规范或其他正式规定文档所需具有的条件或职能。
- (3) 一种反映上面(1)或(2)所描述的条件或职能的文档说明。

IEEE 公布的定义包括了从用户角度(系统的外部行为)和从开发者角度(一些内部特

性)对需求的阐述。关键的问题是一定要编写需求文档。在国内许多开发人员和分析人员都不喜欢编写需求规格说明书,而用户也是既不关心需求规格说明书是否已经完成,也不关心需求规格说明书是否恰当地表述了用户的需求。大家所关心的仅仅是系统何时能够做出来,而这最终导致的一个结果是系统不断被修改,各种 Bug 不断出现,好像没有个尽头。其实任何文档形式的需求都只是一个模型或一种描述^[225],我们要保证所有的风险承担者(Stakeholder)在描述需求的那些名词上能够达成一致的意見,这是尽可能减少需求的二义性、降低需求可能出现变动风险的一个关键。

1. 需求的层次

软件需求包括三个不同的层次——业务需求、用户需求和功能需求,也包括非功能需求。业务需求(Business Requirement)反映了组织机构或客户对系统、产品高层次的目标要求,它们在项目视图与范围文档中予以说明。用户需求(User Requirement)文档描述了用户使用产品必须要完成的任务,这在用例(Use Case)文档或情景脚本(Scenario)说明中予以说明。功能需求(Functional Requirement)定义了开发人员必须实现的软件功能,使得用户能完成他们的任务,从而满足了业务需求。所谓特性(Feature)是指逻辑上相关的功能需求的集合,给用户处理提供能力并满足业务需求。软件需求各组成部分之间的关系如图 14-1 所示。

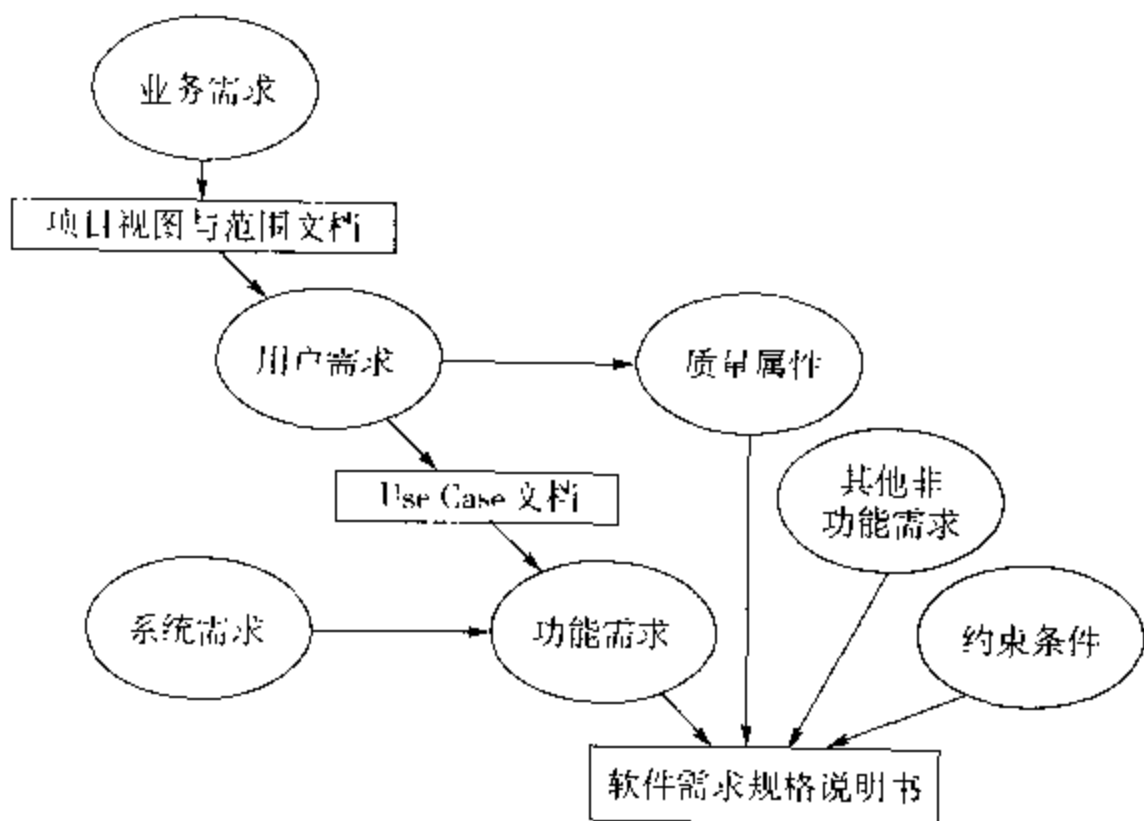


图 14-1 软件需求各层次关系

2. FURPS + 模型

在软件需求规格说明书(Software Requirements Specification, SRS)中说明的需求充分描述了软件系统所应具有的外部行为。软件需求规格说明书在开发、测试、质量保证、项目管理以及相关项目功能中都起了重要的作用。

软件需求规格说明书可以使用 Robert B. Grady 提出的 FURPS + 模型来描述^[226]。FURPS 分别表示:功能性(Functionality)、可用性(Usability)、可靠性(Reliability)、性能

(Performance)、可支持性(Supportability)。+表示:设计约束(Design requirements)、实施需求(Implementation requirements)、接口需求(Interface requirements)和物理需求(Physical requirements)。具体还可以参考[IEEE Std 610.12.1990]。

功能性需求规定了系统无须考虑物理约束而必须能够执行的动作。

功能性需求包括:

- 能力(Capabilities);
- 特性集(Feature Set);
- 安全性(Security)。

可用性需求包括:

- 界面美学(Aesthetics);
- 界面一致性(Consistency);
- 正确性(Correctness);
- 用户文档(Documentation);
- 人员因素(Human Factors);
- 可理解性(Understandability)。

可靠性需求包括:

- 精确性(Accuracy);
- 正确性(Correctness);
- 故障频率/严重性(Frequency/Severity of Failure);
- 可预见性(Predictability);
- 可恢复性(Recoverability)。

性能需求包括:

- 正确性(Correctness);
- 资源消耗(Resource Consumption);
- 响应时间(Response Time);
- 速度(Speed);
- 吞吐量(Thruput)。

可支持性需求包括:

- 可适应性(Adaptability);
- 可兼容性(Compatibility);
- 可配置性(Configurability);
- 正确性(Correctness);
- 可扩展性(Extensibility);
- 可安装性(Installability);
- 可本地化(Localizability);
- 可维护性(Maintainability);
- 可携带性(Portability);
- 可复用性(Reusability);
- 可服务性(Serviceability);

- 可测试性 (Testability)。

设计约束：

设计约束有时也叫设计需求，它规定或约束了系统的设计。

实施需求：

实施需求规定或约束了系统的编码或构建。例如：

- 所需标准；
- 实施语言；
- 数据库完整性策略；
- 资源限制；
- 操作环境。

接口需求：

接口需求规定了：

- 系统必须与之交互操作的外部项；
- 对这种交互操作所使用的格式、时间或其他因素的约束。

物理需求：

物理需求规定了系统必须具备的物理特征。例如：

- 材质；
- 形状；
- 尺寸；
- 重量。

这种需求类型可用来代表硬件要求，如：

物理网络配置需求。

3. 可能的需求风险

不重视需求过程的项目队伍将自食其果。需求工程中的缺陷将给项目成功带来极大风险。这里的“成功”是指推出的产品能够在价格、时间功能、质量上完全满足用户的期望。

常见的需求风险包括：

- 用户缺乏充分的参与

客户经常不明白为什么收集需求和确保需求的质量需花费那么多的功夫，开发人员可能也不重视用户的参与。在某些情况下，与实际使用产品的用户直接接触很困难，而客户也不太明白自己的真正需求。但还是应让具有代表性的用户在项目早期直接参与到开发队伍中，并共同经历整个开发过程。

- 用户需求的不断增加

在开发中若不断地补充需求，项目就会越变越庞大以致超过其计划及预算范围。计划并不总是与项目需求的规模与复杂性、风险、开发生产率及需求变更实际情况相一致，这使得问题更难解决。实际上，问题根源在于用户需求的改变和开发者对新需求所做的修改。

- 模棱两可的需求

模棱两可是需求规格说明书中最为可怕的问题^[226]。它的一层含义是指诸多读者对需

求说明产生了不同的理解；另一层含义是指单个读者能用不止一个方式来解释某个需求说明。模棱两可的需求会使不同的风险承担者产生不同的期望，它会使开发人员为错误问题而浪费时间，并且使测试者与开发者所期望的不一致。

- 不必要的特性

“画蛇添足”是指开发人员力图增加一些“用户可能会欣赏”但需求规格说明书中并未涉及的新功能。经常发生的情况是用户并不认为这些特性很有用，以致在其上耗费的努力“白搭”了。

- 过于简单的规格说明

有时，客户并不明白需求分析有如此重要，于是只作一份简略的规格说明，仅涉及了产品概念上的内容，然后让开发人员在项目进展中去完善，结果可能出现的是开发人员先建立产品的结构之后再完成需求说明。这种方法可能适合于尖端研究性的产品或需求本身就十分灵活的情况^[20]。但在大多数情况下，这会给开发人员带来挫折（使他们在不正确的假设前提和极其有限的指导下工作），也会给客户带来烦恼（他们无法得到他们所设想的产品）。

- 忽略了用户分类

大多数产品是由不同的人使用其不同的特性，每种特性被使用的频繁程度也有所差异，使用者受教育程度和经验水平也不尽相同。如果不能在项目早期就针对所有这些主要用户进行分类，必然导致某些用户对产品感到失望。

- 不准确的计划

对需求分析缺乏理解会导致过分乐观的估计，而当不可避免的超支发生时，会带来颇多麻烦。导致需求过程中软件成本估计极不准确的原因主要有以下5点：频繁的需求变更、遗漏的需求、与用户交流不够、质量低下的需求规格说明和不完善的需求分析（Davis 1995）。

4. 好的需求应具有的特点

一个好的需求应当具有以下特点：

- **完整性。**每一项需求都必须将所要实现的功能描述清楚，以使开发人员获得设计和实现这些功能所需的所有必要信息。
- **正确性。**每一项需求都必须准确地陈述其要开发的功能。
- **一致性。**一致性是指与其他软件需求或高层（系统，业务）需求不相矛盾。
- **可行性。**每一项需求都必须在已知系统与环境的职能和限制范围内可以实施。
- **无二义性。**对所有需求说明的读者都只能有一个明确统一的解释，由于自然语言极易导致二义性，所以尽量把每项需求用简洁明了的用户性的语言表达出来。
- **健壮性。**需求的说明中是否对可能出现的异常进行了分析，并且对这些异常进行了容错处理。
- **必要性。**“必要性”可以理解为每项需求都是用来授权你编写文档的“根源”。要使每项需求都能回溯至某项客户的输入，如 Use Case 或别的来源。
- **可测试性。**每项需求都能通过设计测试用例或其他验证方法来进行测试。
- **可修改性。**每项需求只应在 SRS 中出现一次。这样更改时易于保持一致性。另

外,使用目录表、索引和相互参照列表方法将使软件需求规格说明书更容易修改。

- **可跟踪性。**应能在每项软件需求与它的根源和设计元素、源代码、测试用例之间建立起链接链,这种可跟踪性要求每项需求以一种结构化的、粒度好(fine-grained)的方式编写并单独标明,而不是大段大段的叙述。

另外应当对所有需求分配优先级。如果把所有需求都看作同样重要,那么项目管理者在开发或节省预算或调度中就丧失控制自由度。

14.1.2 测试需求

测试需求的目的是为了验证需求的正确性、完整性、一致性等应具有的特性。需求测试的手段不是很多,目前还处在起步阶段,本章将从评审、测试设计、需求建模测试、原型法测试这几个方面来介绍测试需求的方法,以供感兴趣的同行参考。

14.2 通过评审来测试需求

同行评审是业界公认的最有效的排错手段之一^[228]。在需求测试过程中,使用最多的也是同行评审(Peer Review),尤其是在正规检视(Inspection)中。正规检视是由 Michael Fagan 在 IBM 制定出来的一种非常严格的评审过程^[224]。

需求评审的参与者当中,必须要有用户或用户代表参与,同时还需要包括项目的管理者、系统工程师和相关开发人员、测试人员、市场人员、维护人员等。在项目开始之初就应当确定不同级别、不同类型的评审必须有哪些人员参与,否则,评审可能会遗漏掉某些人员的意见,导致今后不同程度的返工。

14.2.1 需求评审中的常见风险

由于需求的重要性以及描述需求的自然语言的自由性,要求我们必须对需求采取最严格的评审过程。然而在这个过程的实施中,可能会遇到下面这些风险:

- 需求评审的参与者选取不当。需求评审的参与者当中,必须要有用户或用户代表参与,同时还应当包括项目开发相关人员,如开发人员、测试人员、维护人员等。缺乏用户参与的需求评审不能真正代表用户的要求,最终会导致实现的产品不满足用户要求。而缺乏相关人员参与的评审会导致相关人员可能无法及时了解需求情况,尤其是需要的变更,这样会导致相关人员的很多工作需要返工,造成他们进度拖延,最终不同组织之间的关系可能因此紧张。
- 评审规模过大。评审一份几百页的软件需求规格说明书是令人畏惧的。你很可能完全忽略整个评审过程,并继续进行软件的构造开发——这不是一个好的选择。即使是一份中型的软件需求规格说明书,评审人员可能会认真地检查第一部分,一些意志坚定的人可以检查到中间部分,但没有一个人可能检查到最后一部分。

为了避免使评审小组感到不安, 只在需要把软件需求规格说明书作为基线时才进行评审。在评审全部的文档之前或在开发软件需求规格说明书时, 可以采用非正式的、渐增式的评审方式(如走读)。让一些评审人员从文档的不同位置开始检查, 以确保他们认真地检查其中的每一页。如果有足够的审查员, 可以分成几个小组分别审查材料的不同部分。标准推荐的每次需求评审规模大约是 10 ~ 30 页之间。

- 评审小组规模过大。许多项目参与者和客户都与需求有关系, 所以可能要为需求评审的参与者制作一张冗长的名单列表。然而, 庞大的评审小组将导致难于安排会议, 并且在评审会上经常引发题外话, 在许多问题上也难于达成一致意见。一个合理的评审小组规模应当控制在 3 ~ 7 人之间。
- 评审时间跨度太长。一个合理的评审周期应当控制在两个星期以内, 评审周期跨度太大, 会使得评审参与者感觉不到评审的压力, 造成松懈, 最终导致评审效果太差。另外太长的评审周期对项目的进度也是不利的。在评审过程中, 评审会议的时间也不能太长, 太长的评审会议时间会使人疲惫, 并且使人产生畏惧评审的心态, 导致大家不愿意参加评审。同时评审时间太长, 会造成效率低下, 更多地纠缠于一些细节的争吵当中, 因此评审组织者需要适当地控制评审时间和评审秩序, 一个合理的评审会议时间应当控制在 2 个小时以内。

14.2.2 需求评审检查表

需求评审检查表是需求问题检查最好的一个工具。表 14-1 给出了需求评审的一个检查表示例。

表 14-1 需求评审检查表

需求特性	检查内容
清晰性/无二义性	所有定义、实现方法是否清楚地表达了用户的原要求
	在功能实现过程、方法和技术要求的描述上, 是否背离了功能的实际要求
	是否有不能理解或造成误解的描述
完整性	是否有一个内容表格, 该表格包含了所有需求描述
	是否所有的图形、表格都被进行了标号
	是否所有的需求项都被进行了标号, 并提供了索引
	是否有需求可以被定义的更细致, 或更简单
	对于不清晰的信息是否进行了指出
	是否存在有需求让你觉得不舒服
	是否所有与需求相关的设计约束都被包含了
	是否所有与需求相关的性能都被包含了
	是否所有与需求相关的属性都被包含了

续表

需求特性	检查内容
完整性	是否所有与需求相关的外部接口都被包含了
	是否所有与需求相关的通信都被包含了
	是否所有与需求相关的硬件都被包含了
	是否所有与需求相关的数据库都被包含了
	是否所有与需求相关的软件都被包含了
	是否所有与需求相关的硬件都被包含了
	是否所有与需求相关的输入和输出都被包含了
	是否所有与需求相关的安装特性都被包含了
	是否所有与需求相关的维护特性都被包含了
	是否所有与需求相关的安全特性都被包含了
	所有对其他需求的内部交叉引用是否正确
	所有需求的编写在细节上是否都一致或者合适
	需求是否能为设计提供足够的基础
	是否包括了每个需求的实现优先级
	需求定义是否包含了有关文件(指质量手册、质量计划以及其他有关文件)中所规定的需求定义所应该包含的所有内容
	需求定义是否包含了有关功能、性能、限制、目标、质量等方面的所有需求
	功能性需求是否覆盖了所有非正常情况的处理
	是否对各种操作模式(如正常、非正常、有干扰等)下的环境条件都作了规定
	是否对所有功能与时间因素有关的方面都作了考虑
	是否标识出了所有与时间因素有关的功能? 它们的时间准则是否都说明了? 时间准则的最大、最小执行时间是否都定义了
	是否标识并定义了在未来可能会变化的需求
	是否定义了系统所有的输入
	是否标识清楚了系统输入的来源
	是否标识出了系统的输出
	是否说明了系统输入、输出的类型
	是否说明了系统输入、输出的值域、单位、格式等
	是否说明了如何进行系统输入的合法性检查
	是否定义了系统输入、输出的精度
	是否定义了系统性能的各个方面
	在不同负载情况下, 是否规定了系统的生产率

续表

需求特性	检查内容
完整性	在不同情况下,是否规定了系统的响应时间
	是否充分定义了关于人机界面的需求
	是否对需求定义进行了可行性分析和相关文件(资料)是否已归档
	是否对影响需求实现的因素进行了调查,调查结果是否已归档
	是否有商业行为,分析结果是否已归档
	是否详细描述了有关硬件、软件、操作人员、操作过程等方面的安全性
	是否评估了本项目对用户、其他系统、环境的影响特性
	是否按完成时间、重要性对系统功能、外部接口、性能进行了优先排序
兼容性	界面需求是否使软硬件系统具有兼容性
	需求定义的文档是否满足项目文档编写标准?在矛盾时,是否有适当的标准可供选择
一致性	各个需求之间是否一致?是否有冲突和矛盾
	所规定的模型、算法和数值方法是否相容
	是否使用了标准的术语和定义形式
	需求是否与其软硬件操作环境相容
	是否说明了软件对其系统和环境的影响
	是否说明了环境对软件的影响
	所采用的技术是否与用户要求的技术一致
正确性	需求定义是否满足标准的要求
	算法和规则是否有科技文献或其他文献作为基础
	是否定义了对在错误、危险分析中所标识出的各种故障模式和错误类型所需的反应
	是否参照了有关的标准
	是否对每一个需求都给出了理由?理由是否充分
	对设计和实现的限制是否都有论证
可行性	需求定义是否使软件的设计、实现、操作和维护都可行
	所规定的模型、数值方法和算法是否对待解决问题合适?是否能够在相应的限制条件下实现
	是否能够达到关于质量的要求
易修改性	对需求定义的描述是否易于修改(如是否采用良好的结构和交叉引用表等)
	是否有冗余的信息?是否一个需求被定义了多次
健壮性	是否有容错的需求

续表

需求特性	检查内容
易跟踪性	是否每个需求都具有惟一性并且可以正确地识别它
	是否可从上一阶段的文档中找到需求定义中的相应内容
	需求定义是否明确地表明前阶段中提出的有关需求和设计限制都被覆盖了
	需求定义是否便于向后继开发阶段查找信息
可理解性	最终产品的每个特性是否始终用同一个术语进行了描述
	是否每一个需求都只有一种解释
	功能性需求是否以模块方式描述的? 是否明确地标识出了其功能
	是否有术语定义一览表
	是否使用了形式化或半形式化的语言
	语言是否有歧义性
	需求定义中是否只包含了必须的实现细节而不包含不必要的实现细节? 是否过分细致了
	需求定义是否足够清楚和明确使其能够作为开发设计规约和功能性测试数据的基础
	需求定义的描述是否将对程序的需求和所提供的其他信息分离开来了
可测试性	需求是否可以验证(即是否可以检验软件是否满足了需求)
	是否对每一个需求都指定了验证过程
	数学函数的定义是否使用了精确定义的语法和语义符号
性能	是否精确地描述了所有的性能需求和可容忍的性能降低程度? 对每一个性能应包含两方面的内容: a. 在最坏情况的执行结果 b. 本性能失效后, 对系统产生的影响
	是否指定了所有期望的处理时间
	是否指定了数据传输速率
	是否指定了系统的吞吐量
功能	是否清楚、明确地描述了所有的功能
	所有已描述的功能是否是必须的? 是否能满足任务书或系统目标的要求
接口	是否清楚地定义了所有的外部接口
	是否清楚地定义了所有的内部接口
	所有接口是否必须? 各接口间的关系是否一致、正确
数据	在某异常数据(如条件、标志等)下, 是否有尚未考虑到的结果
	对异常数据产生的结果是否作了精确的描述
硬件	是否指定了最小内存需求
	是否指定了最小存储空间需求

续表

需求特性	检查内容
硬件	是否指定了最大内存需求
	是否指定了最大存储空间需求
软件	是否指定了需要的软件环境/操作系统
	是否指定了需要的所有软件设施
	是否指定了所有要与系统一起允许的购买的软件产品
通信	是否指定了目标网络
	是否指定了需要网络协议
	是否指定了需要的网络能力
	是否指定了需要的/估计的网络吞吐量
	是否指定了估计的网络连接数量
	是否指定了最小网络性能需求
	是否指定了乐观的网络性能需求
可维护性	需求定义中是否包括了可行的系统维护方法
	模块或子程序(过程)间的关系是否是松耦合的(即能否保证在对某部分修改后,产生最小的连锁效应)
可靠性	是否为每个需求指定了软件失效的结果
	是否指定了特定失效的保护信息
	是否指定了特定的错误检测策略
	是否指定了错误纠正策略
其他	是否所有的需求都是名副其实的需求而不是设计或实现方案
	是否确定了对时间要求很高的功能并且定义了它们的时间标准
	是否已经明确地阐述了国际化问题
	使用实例是否是独立的分散任务
	使用实例的目标或价值度量是否明确
	使用实例给操作者带来的益处是否明确
	使用实例是否处于抽象级别上,而不具有详细的情节
	使用实例中是否不包含设计和实现的细节
	是否记录了所有可能的可选过程
	是否记录了所有可能的例外条件
	是否存在一些普通的动作序列可以分解成独立的使用实例
	是否简明书写、无二义性和完整地记录了每个过程的对话
	使用实例中的每个操作和步骤是否都与所执行的任务相关
	使用实例中定义的每个过程是否都可行
	使用实例中定义的每个过程是否都可验证

14.3 通过用例设计来测试需求

从V模型上可以知道,验收测试是以系统需求为基础的,系统测试是以功能测试为基础。每个开发阶段的活动都与相应的测试活动是并行进行的。在需求阶段进行系统(验收)测试计划 and 设计,除了能指导最终的系统测试和验收测试执行外,其本身也是对需求的一个验证过程。

通过阅读软件需求规格说明书,通常很难想象在特定环境下的系统行为。以功能需求为基础或者从使用实例派生出来的测试用例可以使项目参与者看清系统的行为。虽然没有在运行系统上执行测试用例,但是设计测试用例的简单动作可以解释需求的许多问题^[26]。如果在部分需求稳定时就开始开发测试用例,那么就可以及早发现问题并以较少的费用解决它们。

设计概念性测试用例可以发现需求的错误、二义性、不可测性、遗漏等方面的问题,为了获得最大的效果,要求测试人员能够独立地去对需求进行思维,从一个不同于开发的角度上进行分析,这可能会是一个逆向的思维过程,在这个过程中,测试人员可能会设计出不同于需求的测试用例,而这最终可能会有两个解释:

- 需求不完整或者需求有错;
- 遗漏了测试用例或者测试用例本身有错误。

不管是哪种解释,最终肯定会提高整个系统的质量。但这个质量的获得是通过冗余的人员来完成的,即:开发人员在系统需求进行进一步分析时,有一组独立的测试人员也在对系统需求进行独立的思维,并从中获取测试用例。尽管这两种思维可能会出现重复,但由于思维的方式不同,最终肯定会使需求变得更清晰和更完善。

一个示例

在此使用 Karl E. Wiegers 给出的一个化学容器的例子来进行说明^[207]。表 14-2 给出了需求的一个描述。

对于这一个需求,开发人员可能会进一步分析得到类似图 14-2 的一个实例对话图。

表 14-2 一个需求示例

业务需求编号	SR-01
业务需求描述	“化学制品跟踪系统”通过鼓励重复使用公司中可用的那些化学制品容器以降低购买费用
操作方式	请求者通过输入化学制品的 ID 号或从化学制品绘图工具导入(Import)化学结构来请求一种化学制品。系统则通过向请求者提供来自化学制品仓库的一个新的或已用过的化学制品容器或者让请求者向外部供应商发送订单,从而满足请求者的要求

续表

功能需求编号	SR-01-001
需求分解为设计需求参数范围(规格)	功能 1: 如果请求化学制品仓库中的容器, 系统将显示可用容器的列表, 用户就可以选择一个容器或要求向外部供应商订购一个新容器 功能 2: (略) 环境: (略) 性能: (略) 健壮性(鲁棒性): (略) 可靠性: (略) 安全性: (略)

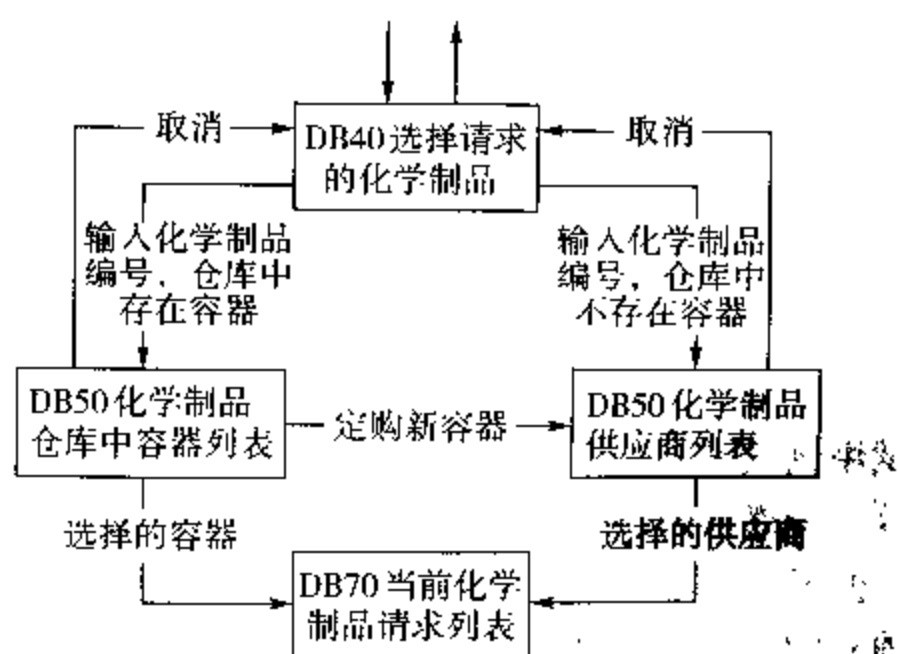


图 14-2 化学制品请求对话图

而同样对于需求表 14-2, 资深的测试人员可能会设计如表 14-3 的测试用例。

表 14-3 化学制品请求测试用例

测试用例编号	输入	操作步骤	期望结果
TC-001	仓库中有的化学制品, 其 ID = 00001	1. 打开选择化学制品对话框 2. 输入化学制品 ID 号 00001 3. 在弹出的仓库容器列表中选择第一个容器 00001-01	显示请求到的化学制品信息
TC-002	仓库中有的化学制品, 其 ID = 00001	1. 打开选择化学制品对话框 2. 输入化学制品 ID 号 00001 3. 在弹出的仓库容器列表窗口中单击“取消”按钮	返回选择化学制品对话框
TC-003	仓库中没有的化学制品, 其 ID = 00002, 并且数据库中有相应的供应商列表	1. 打开选择化学制品对话框 2. 输入化学制品 ID 号 00002 3. 在弹出的供应商列表中选择第一个供应商 00002-O-01	显示订购到的化学制品信息

(1) 系统能够反馈异常信息;

(2) 如果系统不对该异常处理, 至少系统要能够容忍这个错误, 而不会造成系统崩溃或功能丧失。

不管怎么处理, 相应的要求必须清晰地描述到需求列表 14-2 中。一个可能的最终需求列表可以用表 14-4 来表示。

表 14-4 更新后的需求列表

业务需求编号	SR-01		
业务需求描述	“化学制品跟踪系统”通过鼓励重复使用公司中可用的那些化学制品容器以降低购买费用		
操作方式	请求者通过输入化学制品的 ID 号或从化学制品绘图工具导入(Import)化学结构来请求一种化学制品。系统则通过向请求者提供来自化学制品仓库的一个新的或已用过的化学制品容器或者让请求者向外部供应商发送订单, 从而满足请求者的要求		
功能需求编号	SR-01-001	SR-01-002	SR-01-003
需求分解为设计需求参数范围(规格)	功能 1: 如果请求化学制品仓库中的容器, 且仓库中有相应的容器, 系统将显示可用容器的列表, 用户可以选择一个容器	功能 2: 如果请求化学制品仓库中的容器, 且仓库中没有相应的容器, 系统将显示可用供应商列表, 用户可以选择向供应商订购一个新容器 环境: (略) 性能: (略)	健壮性(鲁棒性): 如果用户输入的化学制品 ID 号非法, 或该制品无法找到仓库中对应的容器并且也无法找到对应供应商信息, 系统不能出现崩溃或功能丧失, 并且需要反馈给用户出错信息 可靠性: (略) 安全性: (略)

在上面的例子中, 分析员和测试人员在编写代码以前把需求、分析模型和测试用例结合在一起检测遗漏、错误和不必要的需求。软件需求在概念上的测试是通过在开发早期的阶段寻找需求错误, 从而成为一种在控制项目费用和进度上的强有力的技术。

14.4 需求建模测试

需求的建模包括把需求转换成图形模型或形式化语言模型。需求的图形化分析模型包括数据流图(Data Flow Diagram, DFD)、实体关系图(Entity-Relationship Diagram, ERD)、状态转化图(State-Transition Diagram, STD)、对话图(Dialog Map)和类图(Class Diagram)。这些图形化模型一般都需要借助一定的 CASE(Computer-Aided Software Engineering)工具。本节介绍的方法主要借助于自动化分析工具本身提供的检测手段来对需求进行测试, 而这类检测主要提供描述上的完整性检查, 需求项之间的不一致性检查等方面的功能。同时, 使用这类自动分析工具有助于获得需求的质量特性, 包括: 有效性、一致性、可靠性、可存活性、可用性、正确性、可维护性、可测试性、可扩展性、可交互性、可重用性、可携带性等。

14.4.1 统一建模语言

统一建模语言(Unified Modeling Language, UML)提供了一种描述需求 and 设计方面的统一符号,它不仅统一了Booch、Rumbaugh和Jacobson的表示方法,而且对其作了进一步的发展,并最终统一为大众所接受的标准建模语言^[229]。

UML是一种定义良好、易于表达、功能强大且普遍适用的建模语言。它融入了软件工程领域的新思想、新方法和新技术。它的作用域不限于支持面向对象的分析与设计,而是支持从需求分析开始的软件开发的全过程。

作为一种建模语言,UML的定义包括UML语义和UML表示法两个部分。

- UML语义 描述基于UML的精确元模型定义。元模型为UML的所有元素在语法和语义上提供了简单、一致、通用的定义性说明,使开发者能在语义上取得一致,消除了因人而异的表达方法所造成的影响。此外UML还支持对元模型的扩展定义。
- UML表示法 定义UML符号的表示法,为开发者或开发工具使用这些图形符号和文本语法作为系统建模提供了标准。这些图形符号和文字所表达的是应用级的模型,在语义上它是UML元模型的实例。

标准建模语言UML的主要内容可以由下列5类图(共9种图形)来定义:

- 第一类是用例图(Use Case Diagram),包括用例(Use Case)和角色(Actor),用来从用户角度描述系统功能,并指出各功能的操作者。
- 第二类是静态图(Static Diagram),包括类图、对象图和包图。其中类图描述系统中类的静态结构。不仅定义系统中的类,表示类之间的联系如关联、依赖、聚合等,也包括类的内部结构(类的属性和操作)。类图描述的是一种静态关系,在系统的整个生命周期都是有效的,它表达了系统在一个稳态必须满足的对象间的关系。对象图是类图的实例,几乎使用与类图完全相同的标识。它们的不同点在于对象图显示类的多个对象实例,而不是实际的类。一个对象图是类图的一个实例。由于对象存在生命周期,因此对象图只能在系统某一段时间存在。包图由包或类组成,表示包与包之间的关系。包图用于描述系统的分层结构。
- 第三类是行为图(Behavior Diagram),包括状态图、活动图,用来描述系统的动态模型和组成对象间的交互关系。其中状态图描述类的对象所有可能的状态,以及事件发生时状态的转移条件。通常,状态图是对类图的补充。而实际上并不需要为所有的类画状态图,仅为那些有多个状态其行为受外界环境的影响而发生改变类画状态图。而活动图描述满足用例要求所要进行的活动以及活动间的约束关系,有利于识别和表达并行活动。
- 第四类是交互图(Interactive Diagram),包括顺序图、合作图。用来描述对象间的交互关系。其中顺序图显示对象之间的动态合作关系,它强调对象之间消息发送的顺序,同时显示对象之间的交互。合作图描述对象间的协作关系,合作图跟顺序图相似,显示对象间的动态合作关系。除显示信息交换外,合作图还显示对象以及它们之间的关系。如果强调时间和顺序,则使用顺序图;如果强调上下级关

系,则选择合作图。这两种图合称为交互图。

- 第五类是实现图(Implementation Diagram)。包括构件图、配置图。其中构件图描述代码部件的结构及各部件之间的依赖关系。一个部件可能是一个资源代码部件、一个二进制部件或一个可执行部件。它包含逻辑类或实现类的有关信息。部件图有助于分析和理解部件之间的相互影响程度,配置图定义系统中软硬件的物理体系结构。它可以显示实际的计算机和设备(用节点表示)以及它们之间的连接关系,也可显示连接的类型及部件之间的依赖性。在节点内部,放置可执行部件和对象以显示节点跟可执行软件单元的对应关系。

1. Use Case 图

在软件开发的需求阶段,主要使用到的图形是 Use Case 图。实际上,人们在进行软件开发时,无论是采用面向对象的方法还是传统方法,首先要做的就是了解需求。在实践中,分析典型的用例是开发者准确迅速地了解用户要求和相关概念的最常用也最有效的方法,是用户与开发者一起深入剖析系统功能需求的起点。但是,过去人们只是在不自觉地这样工作,而且分析用例的方法也很不规范,通常只是口头的交流,或者在随手找到的纸上画一些潦草的示意图,符号简单、含义模糊、只有少数人心领神会,更谈不上建立正式的文档。Ivar Jacobson 首先提出了用例分析方法,并因此闻名于世。一般认为,引入用例的概念并用于开发需求,是面向对象技术进入第二代的标志。

Use Case 是 UML 的核心,贯穿了 RUP 开发方法的整个过程,实际上 RUP 就是一种 Use Case 驱动的开发方法^[309]。在传统的 OO 系统模型中,很难明确地说明系统是怎样完成其目标的,这是由于系统在执行特定的任务时,没有一条主线。在 UML 中因为 Use Case 定义了系统的行为,它就是一条这样的主线。

从本质上讲,一个 Use Case 是用户与计算机之间的一次典型交互作用。每个 Use Case 在外部 Actor 和目标系统之间定义了一个面向目标的接口集。Actor 是系统外部与系统交互的部分^[229]。一个 Actor 可以是一类用户、用户扮演的角色,或者其他系统。Cockburn 区分了主要角色(Primary Actor)和次要角色(Secondary Actor)^[232]。主要角色是指需要系统提供帮助的人或物。次要角色是指系统需要从该角色获取帮助的人或物。

一个 Use Case 最初由一个带有特定目的的用户发起,并且在该目标满足后成功结束。它描述了 Actor 与系统之间必需的交互顺序以满足特定的目标服务。它还包括了可能的顺序变化,例如,一个可替换的同时也能满足目标的顺序,以及可能导致失败的顺序来结束服务。出现这些情况可能是由异常的行为、错误处理等引起的。在这里,系统被看作是一个黑盒(Black Box),与系统的交互,以及系统的反应都可以在系统外部被观察到。这样 Use Case 捕获到了谁(Who)对系统做了什么(What),是为了什么目的(Goal)。而对系统的内部行为没有进行处理。一个完整的 Use Case 集合应当包括所有使用系统的不同方法集合,并因此定义所有系统需要的行为,划分系统的边界。

通常 Use Case 步骤使用易于理解的结构化语言进行描写。这使得用户可以很容易地理解和验证 Use Case,并且可以鼓励用户参与到需求定义当中去。

在 Use Case 中使用到了场景 (Scenarios) 这个概念。场景是一个 Use Case 的实例 (Instance)，代表通过 Use Case 的一个单一路径。这样，就可以构造一个场景用于通过 Use Case 的主流程，以及其他一些场景用于每个可能的变化流程。场景可以被描述成顺序图 (Sequence Diagrams)。

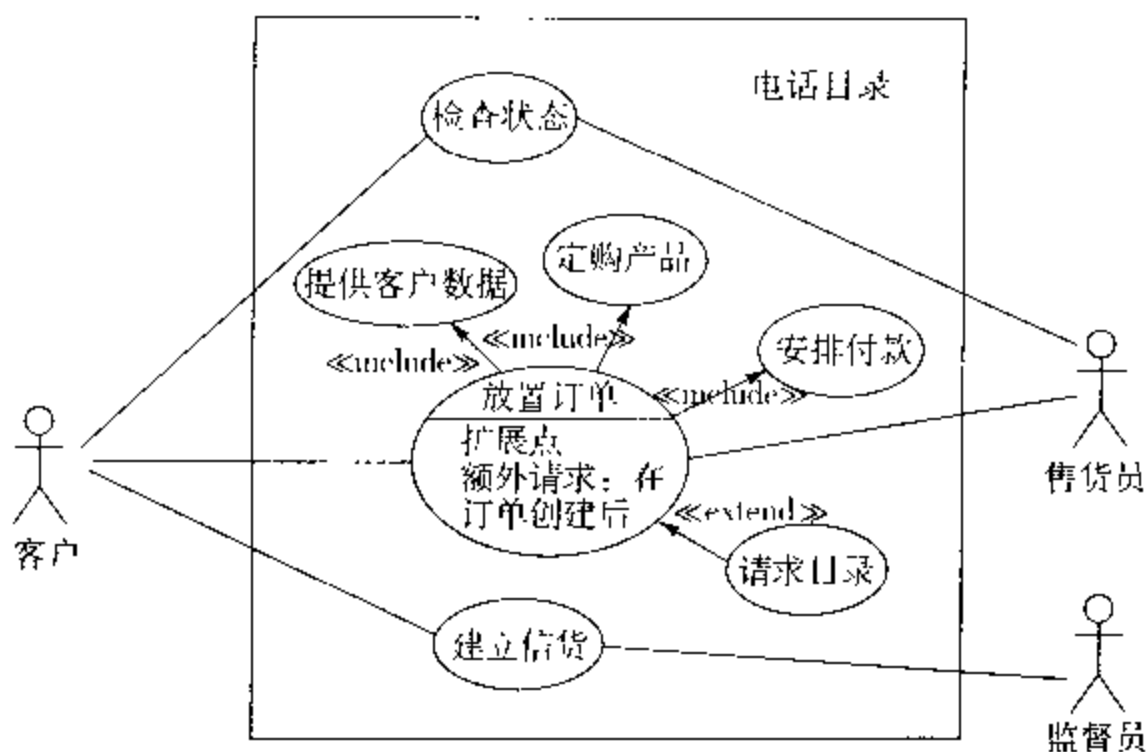
一个系统可能会包含许多 Use Case。这些 Use Case 相互之间的关系如何进行表达呢？UML 提供了三种表示关系用于表示 Use Case 之间的关系。它们分别是：归纳 (Generalization，有些资料把它翻译成“泛化”)，包含 (Include，有些资料把翻译成“使用”) 和扩展 (Extend)。

两个 Use Case 之间的包含关系表示在被包含的 Use Case 中的行为序列被包含在基础 Use Case 行为序列中。这样，包含一个 Use Case 类似于调用一个子过程的概念。

扩展关系提供了一种方法用于捕获一个 Use Case 的变化。扩展不是一个真正的 Use Case，而是改变步骤到一个已经存在的 Use Case 中。典型的扩展被用于指定处理假设为“假”时出现的步骤变化。扩展关系包括扩展发生必须满足的条件，以及在基础类中定义的扩展点位置。

Use Case 之间的归纳关系隐含了“子 Use Case 包含的所有属性，行为顺序和在父 Use Case 中定义的扩展点，并且参与到所有父 Use Case 的关系中”^[229]。子 Use Case 可以定义新的行为序列，同时加入行为到并且特化父 Use Case 已存在的行为。

图 14-4 给出了一个 Use Case 图的例子，表 14-5 给出了 Use Case 的一个模板，表 14-6 给出了一个 Use Case 扩展的模板^[233]。



2. Use Case 测试

既然可以使用 Use Case 来表示用户的需求，并且 Use Case 避免了自然语言描述需求的二义性，可以自由地在不同的用户之间传递信息，那么在需求测试时，重点就落在了如何测试 Use Case 上了。

表 14-5 Use Case 模板

用例(Use Case)	Use Case 标识符和引用序号以及修改历史
描述(Description)	描述 Use Case 要获得的目标以及需求来源
角色(Actors)	Use Case 涉及到的所有 Actors
假设(Assumptions)	Use Case 成功结束必须为真的条件
步骤(Steps)	为了获得 Use Case 目标, Actor 必须与系统交互的操作步骤
变量(Variations)(可选)	Use Case 步骤中使用的所有变量
非功能(Non-Function)	Use Case 必须满足的任何非功能需求
问题(Issues)	还没有被解决的问题列表

表 14-6 Use Case 扩展模板

用例扩展(Use Case Extension)	<扩展标识符> 扩展 <用例标识符>
变化(Change)	通过扩展要获得的目标
条件(Condition)	扩展发生必须满足的条件
步骤(Steps)	用新的或可替换的步骤来描述变化, 这些步骤被应用到扩展点上的 Use Case 上
变量(Variations)(可选)	扩展 Use Case 步骤中使用的所有变量
非功能(Non-Function)	扩展 Use Case 必须满足的任何非功能需求
问题(Issues)	还没有被解决的问题列表

测试 Use Case 的方法有两种:

- 使用 Use Case 建模工具, 例如 Rational Rose, 这类工具本身具有检查 Use Case 的功能, 包括语法的正确性、检查是否完整、是否一致等。但是这类工具在检查需求的遗漏或需求本身描述错误方面都比较弱, 因此更好的测试方法是使用第二种方法;
- 使用情景测试(Scenarios Testing)。在情景测试中, 使用角色扮演的方法, 在该方法中给每个项目组成员分配一个角色, 角色可以是用户、系统本身、其他系统, 有时是系统维护的实体。然后, 小组对 Use Case 的每一个情景进行走读, 演示使用系统的方法。在此过程中, 将讨论谁负责什么事情, 对每个角色的职责进行记录。让系统分析员扮演用户或客户的角色有助于真正地了解问题所在。另外, 在情景测试过程中, 让用户充分参与有助于发现一些遗漏或错误的需求。在第 15 章关于构架设计(Architecture Design)的评审方法中也用到了情景测试方法, 可以一起进行参考。

14.4.2 消息顺序图(MSC)

在实时系统中, 其需求的描述主要是关于协议的一个说明。在电信行业, 用得最多的需求描述语言是 MSC(Message Sequence Charts)图。MSC 描述了许多独立的消息传送实体之间的交互, 它是 ITU-T 推荐使用的方法, 其标准包含在 Z.120 中。与之对应推荐使用的设计语言是 SDL, 关于 SDL, 将在 15 章进行介绍。

MSC 是一种比 Use Case 更形式化的表示语言, 在 MSC 图中包含实例和消息两个主体。实例由实例头(一个空心的方框, 方框中是实例的名字)、实例轴(一条垂直线, 表示时间轴)、实例结尾(一个黑色/灰色实体方框)三部分组成; 消息包含了输出事件、输入事件和任选参数三个部分。在 MSC 中, 还用到了定时器消息, 并且可以使用一些简单的表达式, 以表示顺序执行情况(seq)、替换执行情况(alt)、循环(loop[.]), 并行执行(par)、任选执行(opt)和异常情况(exc)。关于 MSC 的详细语法请参考 Z.120 标准, 这里就不赘述了。图 14-5 是一个带有循环的 MSC 图例。

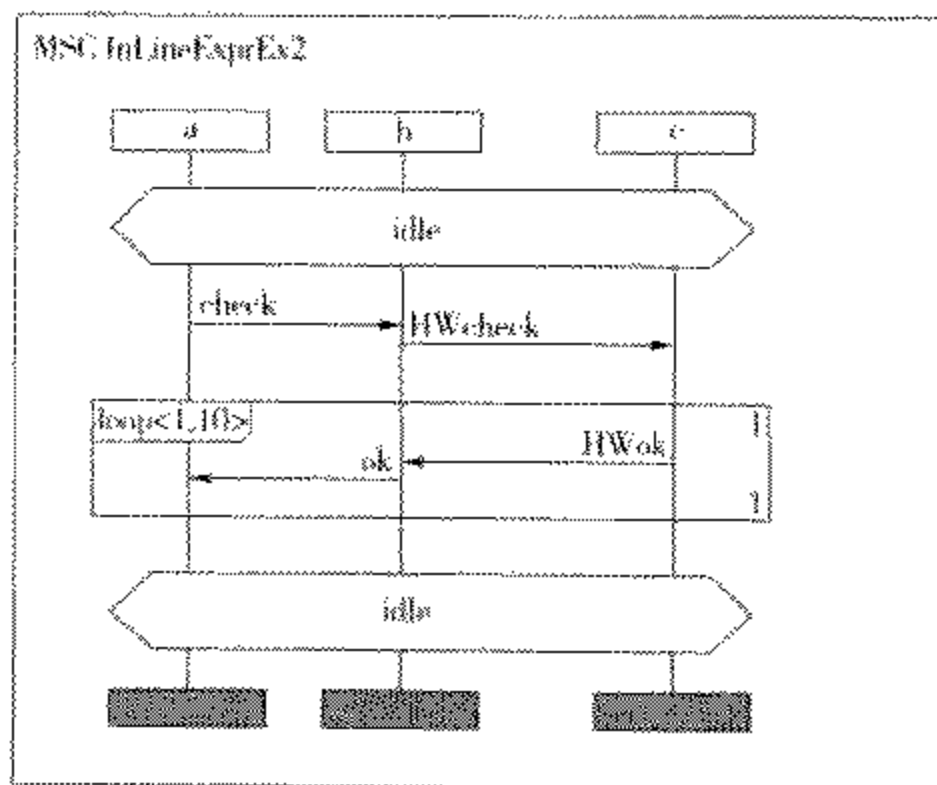


图 14-5 MSC 图示例

一个系统可能会有很多 MSC 图, 为了把这些 MSC 图综合起来, 可以使用 HMSC(High Message Sequence Charts)。HMSC 用于描述 MSC 的组合, 包括 MSC 的启动、停止、状态、连接点及 MSC 的引用等关系。图 14-6 是一个 HMSC 的示例。

MSC 图测试

MSC 对测试的作用是双向的。一方面, 可以借助 MSC 图来设计正向用例、反向用例和异常用例, 包括自动从 MSC 图中导出测试用例。另一方面, 可以通过设计测试用例的手段来发现 MSC 图中存在的问题, 类似于 14.3 节描述的方法。

从消息的交互过程上, 也可以借助 14.4.1 节提出的情景测试方法对 MSC 图进行测试。

由于 MSC 图使用了严格的形式化语言描述, 因此 MSC 图本身的一些语法验证和部分语义验证还可以借助 CASE 工具本身, 例如 Telelogic 公司的 SDT 就是这种工具。

14.4.3 分析建模工具介绍

在进行需求建模时, 最好能够借助于 CASE 工具。那么有哪些 CASE 工具能够帮助我们进行需求分析和需求测试呢?

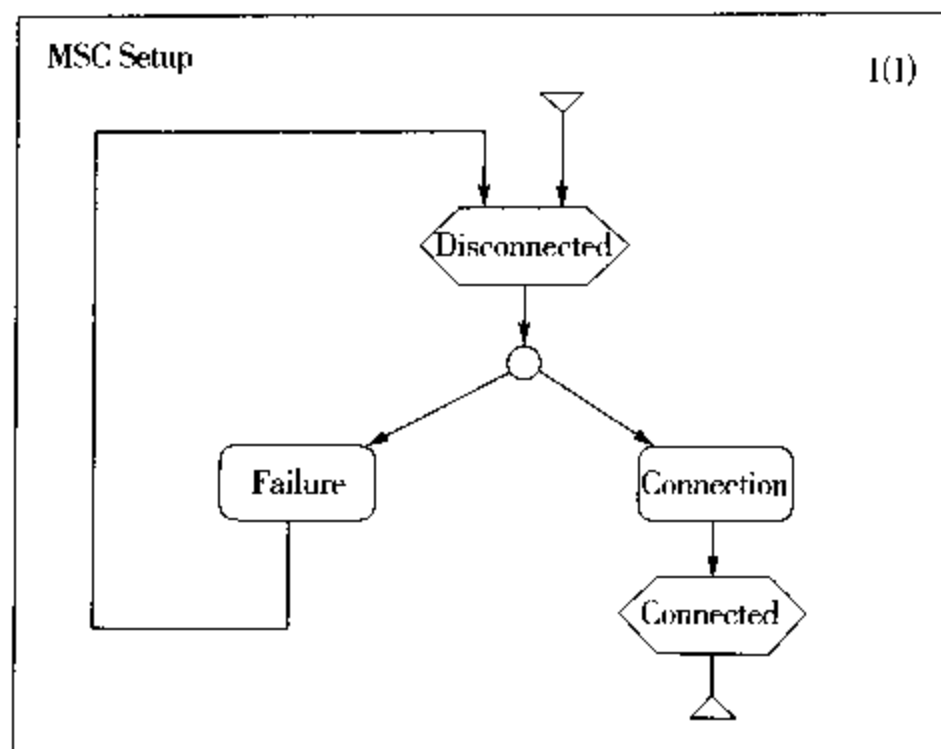


图 14-6 HMSC 示例

STSC(Software Technology Support Center)是一个专门研究软件技术、工具的组织,并且经常为美国空军,国防部提供技术支持。该组织每年都会出一份“需求工程和技术报告”^[234],在该报告中,详细地列出了市场上可用的各种分析设计 CASE 工具,在该报告中,把这类工具定义为 Upper CASE 工具,Upper 的意思是这些工具是使用在软件开发生命周期早期阶段。该报告对工具的分类主要按照以下几个方面进行:

- 生命周期阶段。在这里主要是需求阶段和高层设计阶段;
- 通用方法。主要包括 3 种重要的方法:面向对象方法、面向过程方法、面向行为方法;
- 针对的应用。嵌入式、C³I(Communications, Command, Control, and Intelligence)、商用嵌入式、科学/工程/技术、并行或分布式、人工智能、商用(类似 MIS 系统等);
- 功能能力;
- 产品质量。

表 14-7 给出了一个工具列表(仅给出了可用于需求分析的列表,关于用于高层设计的工具列表可以参考本书第 15 章相关内容,更详细的内容介绍请参考附录 E 的^[234]),在该列表的“类型”这一列中,各缩写代表的意义如下:

A: 需求规格和分析

D: 设计

C: 编码

M: 维护

T: 测试

E: 环境

U: 文档或管理设施,例如配置管理工具或项目管理工具

O: 其他

“目标”这一列中,各缩写代表的意义如下:

TECH: 技术系统

MIS: MIS 系统

RT: 实时系统

HRT: 高实时系统

ALL: 所有系统

“平台”这一列中, 各缩写代表的意义如下:

DT: 桌面机

WS: 工作站

MF: 主机

表 14-7 可用需求分析工具列表

工具名称	供应商	可用类型	目标应用	支持平台
ADADL	Software Systems Design, Inc.	A, D, C, U	ALL	ALL
Argos	Versant Object Technology	A, D, C, E	MIS	WS, DT
ARIS/DesignGen	Software Systems Design, Inc.	A, D	ALL	WS
Axiom-SA	STG, Inc.	A, U	MIS, TECH, RT	DT
AxiomSys	STG, Inc.	A, U	MIS, TECH, RT	DT
CA-Metrics	Computer Associates	A, U	MIS	DT
DataViews	DataViews Corp	A	RT	MF, WS
DOORS	Zycad Corp	A, U	ALL	DT, WS
EiffelCase	ISE	A, D, E	ALL	WS
ERWIN/ERX	LogicWorks	A, D, E,	MIS, TECH	DT, WS
ES RE/Vision	Eden Systems Corp	A	MIS	DT, MF
Expert/CIO	P-Cube Corp	A	MIS, RT	DT
firstcase	AGS Management Systems, Inc.	A, D, E	MIS, RT	DT
Foresight	Nu Thena Systems	A, D	TECH, RT	WS
Foundation Design	Anderson Consulting Foundation	A, D, E, U	ALL	DT
Foundation Vista	Menlo Business Systems	A, D, U	ALL	DT
IE Advantage	Information Engineering Systems Corp	A, D, E, U	ALL	DT
iRAT	introspect Technologies, Inc.	A, D, E, U	TECH, RT	DT
MacAnalyst	Excel Software	A, D, C,	ALL	DT
MacDesigner		E, U		
McCabe	McCabe and Associates, Inc.	A, D, C,	TECH, RT	ALL
Object-Oriented Tool		E, U		
ObjectMaker,	Mark V Systems	A, D, C,	ALL	DT, WS
ObjectMaker TDK		E, U, O		
ObjectTeam	Cadre Technologies	A, D, C,	TECH, RT	ALL
		U, E		
ObjectTeam for OMT	Cadre Technologies	A, D, C U	ALL	WS, DT

续表

工具名称	供应商	可用类型	目标应用	支持平台
Obsydian	Synon, Inc.	A, D, C, E, U	MIS	DT
Paradigm Plus	ProtoSoft	A, D, C, E, U, O	ALL	DT, WS
ProMod-Plus	G & E Systems	A, D, C, E, U, O	ALL	ALL
ProVision Workbench	Proforma Corp	A, U, E, O	MIS	DT
Rational Rose	Rational Software Corp	A, D, U, E	ALL	WS, DT
Rational Rose/Ada, Rational Rose/C++	Rational Software Corp	A, D, C, U, E	ALL	WS
RDBMS App Dev	Cadre Technologies	A, O	MIS	WS, MF
RTM	Marconi System Technology	A, U	ALL	WS
SES/Workbench	SES	A, D, C, O	ALL	WS
SoftTest	Bender & Associates, Inc.	A, T	ALL	DT, WS
System Engineer	LBMS, Inc.	A, D, C, E, U, O	MIS, TECH	DDT
Teamwork	Cadre Technologies, Inc.	A, D, C, E, U, O	MIS, TECH	MF, MF
Teamwork/ Dynamic Analysis	Cadre Technologies, Inc.	A	TECH, RT	WS, MF
Teamwork/IMSQL	Cadre Technologies, Inc.	A, O	MIS, TECH	WS, MF
Teamwork/RT	Cadre Technologies, Inc.	A, D, U	TECH, RT	WS, MF
Teamwork/SA	Cadre Technologies, Inc.	A, D, U	TECH, RT	WS, MF
TurboCASE	StructSoft, Inc.	A, D, C, O, U	MIS, TECH, RT	DT
TurboCASE/Sys	StructSoft, Inc.	A, U	TECH, RT	DT

14.4.4 需求的形式化描述

把需求描述成形式化规格,可以使需求具有形式化的优点——严格。并且一旦需求被形式化之后,对需求的测试也会变得简单,因为形式化语义本身就具有很强的可验证性。然而这种方法的使用范围非常狭窄,它要求分析人员有深厚的数学基础和形式化语义经验。并且对于一个极其复杂的系统来说,要把其需求完全转成形式化语言的工作量是极其庞大,且繁琐的工作。一般安全级别非常高的软件产品才有可能使用该方法,例如:航天工业或医疗仪器等。在教学研究中,这种方法的研究还在被继续着。图 14-7 是一个来自 NASA 的形式化需求表示样例。

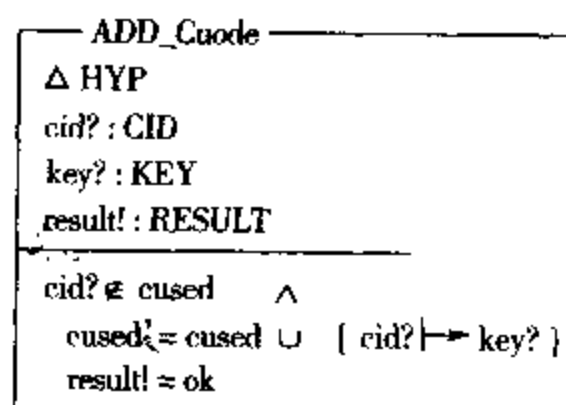


图 14-7 需求的形式化表示样例

14.5 基于原型的测试

人的思维是难以琢磨且易变的，这也是造成需求做不好和需求经常变动的一个根源。尤其在 GUI 软件领域。人们对 GUI 总有各种各样不同的要求，如何获得用户最一般的需求呢？

原型法 (Prototype Method) 是目前业界认为在获取需求和理解需求方面最好的方法之一。原型可以使新产品实在化，为 Use Case 带来生机，并消除你在需求理解上的差异。比起阅读一份冗长无味的软件需求规格说明书，用户通常更愿意尝试建立有趣的原型。

14.5.1 原型的目的

一个软件原型是所提出的新产品的部分实现。使用原型有 3 个主要目的：

- 明确并完善需求。原型作为一种需求工具，它初步实现所理解的系统的一部分。用户对原型的评价可以指出需求中的许多问题，在开发真正产品之前，可以用最低的费用来解决这些问题。
- 探索设计选择方案。原型作为一种设计工具，用它可以探索不同的用户界面技术，使系统达到最佳的可用性，并且可以评价可能的技术方案。
- 发展为最终的产品。原型作为一种构造工具，是产品最初子集的完整功能实现，通过一系列小规模的开发循环，可以完成整个产品的开发。

在本节中我们主要使用到了原型的第一种目的，对于第二种情况一般会应用到设计测试当中，参考第 15 章内容。在需求分析的过程中，原型有助于发现和解决需求中的二义性。二义性和不完整性使开发者对所开发的产品产生困惑，建立一个原型有助于说明和纠正这些不确定性。用户、经理和其他非技术项目风险承担者发现在确定和开发产品时，原型可以使他们的想象更具体化。原型比开发者常用的技术术语更易于理解。

14.5.2 原型的种类

根据原型的实现情况，可以把原型分为“水平原型 (Horizontal Prototype)”和“垂直原型 (Vertical Prototype)”两类。

- 在水平原型中，只实现了与用户交互的界面，而界面背后的细节没有被实现或实现很少。一个水平原型就像一个电影集。它在屏幕上显示出用户界面的正面像，可能允许这些界面之间的一些导航，但是它仅包含少量的功能并没有真正实现所有的功能。有一些导航可能起作用，但是用户可能仅看到描述在那一点将真正显示的内容的信息。数据库查询所响应的信息是假的或者只是一个固定不变的信息，并且报表内容也是固定不变的。虽然原型看起来似乎可以执行一些有意义的工作，但其实不然。这种模拟足以使用户判断是否有遗漏、错误或不必要的功

能。原型代表了开发者对于如何实现一个特定的使用实例的一种观念。用户对原型的评价可以指出使用实例的可选过程、遗漏的过程步骤或原先没有发现的异常情况。

- 在垂直原型中,实现了一部分应用功能。当不能确信所提出的构造软件的方法是否完善或者当需要优化算法、评价一个数据库的图表或测试临界时间需求时,就要开发一个垂直原型。垂直原型通常用在生产运行环境中的生产工具构造,使其结果一目了然(更有意义)。比起在软件的需求开发阶段,垂直原型更常用于软件的设计阶段以减少风险。

根据原型的使用情况,可以把原型分为“抛弃型原型(Throwaway Prototype)”和“演化型原型(Evolutionary Prototype)”。

- 当建立抛弃型原型时,你忽略了很多具体的软件构造技术。而强调在健壮性、可靠性、性能和长期的可维护原则下迅速实现软件并易于维护。基于这一原因,你不能将抛弃型原型中的代码移植到你的产品系统中,除非它达到产品质量代码的标准,否则,你和用户将在软件生存期中遭遇种种麻烦。当遇到需求中的不确定性、二义性、不完整性或含糊性时,最合适的方法是建立抛弃式模型。你需要解决这些问题以减少在继续开发时存在的风险。原型可帮助用户和开发者想象如何实现需求和可以发现需求中的漏洞。它还可以使用户判断出这些需求是否可以完成必要的业务过程。
- 在已经清楚地定义了需求的情况下,进化型原型为开发渐增式产品提供了坚实的构造基础。进化型原型是螺旋式软件开发生存周期模型的一部分,也是一些面向对象软件开发过程的一部分^[235]。与抛弃型原型的快速、粗略的特点相比,进化式模型一开始就必须具有健壮性和产品质量级的代码。因此,对于描述相同的功能,建立进化型原型比建立抛弃型原型所花的时间要多。一个进化型原型必须设计为易于升级和优化的,因此,你必须重视软件系统性和完整性的设计原则。要达到进化型原型的质量要求并没有捷径。

对需求分析来说,我们建议使用水平原型和抛弃型原型。这并不是说要同时使用两种原型,而是同一个原型在两种不同分类方法中的不同归属。

14.5.3 原型的测试方法

在原型的测试上可以使用类似于场景测试的方法(参考14.4.1节)或需求讨论会的方法。不过在这里,不需要进行角色扮演,而是把典型的风险承担者聚集到一起,进行原型的演示,在演示的过程中,可以统一对需求的理解,同时来寻找需求中的歧义、不一致性,以及被丢失的需求。

由于原型是一个已经可以运行的实体,尤其是设计原型(使用垂直原型和演化原型),对于这类原型,可以使用传统的动态测试方法直接对原型进行测试,通过这种动态测试来发现原型中可能隐含的错误。

14.6 本章小结

需求是一个软件开发项目的灵魂所在，有着至关重要的位置。保证需求的质量是一个项目成败的关键。本章，我们从需求评审、用例测试、建模测试和原型测试等多个角度探索对需求的验证，其最终的目的是希望尽可能地使需求稳定，减少项目开发过程中的需求变化，从而加快项目的开发进度，降低项目的开发成本，提高最终软件产品的质量。

收集需求并编写需求文档是软件项目设计成功的很好起点。但还需要保证需求的正确性，使需求能体现出良好需求说明的全部特性。如果能把早期的黑盒子测试设计、非正式需求评审、软件需求规格说明书检视和其他需求验证技术相结合，你将花比以前更少的时间、更低的费用来构造质量更高的系统。

第15章 设计测试

在前一章，分析了如何进行需求测试的一些方法和概念。这一章将讨论如何对设计进行测试。设计测试和需求测试在很多方面具有类似性，一些方法可以相互借鉴。本章涉及到的设计主要指构架设计、高层设计和/或概要设计。希望通过本章的学习可以了解一些有关设计和设计测试方面的知识：

1. 设计具有哪些特点？
2. 为什么要进行构架设计？
3. 常见的软件构架视图有哪几种？
4. 什么是场景？
5. 描述 SAAM、ASAAM 和 ATAM 的策略，并分析它们之间的关系。

15.1 设计测试概述

软件开发过程是一个从概念到实体的逐步细化过程，在这个过程中，构架设计(Architecture Design)或高层设计(High-Level Design)起了关键性的作用，这类设计把抽象的描述性需求转换成可操作的系统构架。在这个阶段，系统已经有了一个比较具体的框架，可以评价其可靠性、稳定性、可扩展性等，包括其符合需求的程度。设计的测试是对系统结构设计进行的一个检测，主要验证设计的质量属性。类似于需求的测试，设计测试也可以从文档评审、模型测试，原型测试或模拟器测试等多个角度进行验证。

15.1.1 什么是设计

尽管一直都存在着关于设计的争议，许多人认为设计不仅仅是一个给定现实世界问题的科学知识的应用^{[236][237][238]}。设计是一个创造性的活动，不能完全简化为标准的步骤，也不应当纯粹地认为是问题的解决。一个设计人员缺乏遵从从一个很好定义的工程原则约束，因为设计本身是混乱的。它包含，但不限于创造性的解决问题的能力。有很多技术可以应用到设计过程中，其中许多技术是明确的、可以通用的，然而，有一些技术是模糊的、无法说明的。Smith 和 Tabor 认为设计既是一门科学，也是一门艺术——它是自发的、无法预计和难以定义的^[240]。

软件设计处在计算机学科的交叉地带：硬件工程和软件工程、编程、人员因素研究、人类工程学。它是人、机器以及与之交互的不同接口(如：物理的、感官的、心理的)的研究^[238]。设计的核心是对话的思想：理论和实际的对话；约束与权衡的对话；设计人员和他掌握材料的对话；设计人员和用户的对话。设计要求在许多经常彼此矛盾的因素之间做出平衡，并且需要广泛的知识。

设计具有以下特点:

- 设计是有目的的,故意的行为。设计关心的是创造人为的形式,它细化了人的目标、渴望和想法。设计的产生是因为我们要寻找满足一些现实的、技术的目标。这些目标可以是非常具体的,例如造一座桥,也可以是非常抽象的,例如写一首诗。设计有着内在的目的论。我们必须了解设计同时被生理和心理所控制。设计无法遵从任何正式的、一致的或系统的方法。设计是直觉,模糊知识和内部反应。
- 设计必须以人为中心。Simon 认为任何事务都处在内部环境和外部环境的交接处。外部环境是一个物理环境,是事务所存在并且必须遵从其自然的物理规则。而内部环境是其操作的内在环境^[241]。在绝大多数情况下,外部环境包括了人以及与之交互的其他物品。而对于软件设计来说,这更为复杂。因为我们所考虑的物品并不是自然界存在的有形实体。计算机具有物理不透明性,其内部工作情况对我们是隐藏的。然而,计算机同时又是功能透明的,当其被正确操作时,其行为是可被检测的。任何软件的人的因素通常被认为是它的接口。现在,GUI 已经变得非常普遍。GUI 所确认的问题通常隐藏了软件内部的工作。这是用户思维时所依赖的一个接口,完全不同于机器内部所实际进行的工作。
- 设计是一个知识强化的行为。设计人员把彼此相关的众多知识体进行了综合,动态地把知识转换成设计任务。在这些知识中,有些完全不是特殊技能,而是一些通用的知识,不过设计人员可以比别的人更熟练地应用这些知识。设计人员同时还需要领域内的科学知识。
- 设计具有历史依赖性。历史给设计活动提供了一条或另一条路径。任何产品,即使是最创新的想法,都是过去历史的一个演进。
- 设计是有选择的。当一个产品被创建时,设计人员对其内部和外部环境的了解和领悟是不可避免的,并且是有选择性的。这有很多原因,部分是因为内心的冲突,部分是因为不得不使用对相关环境知识的状态。这个选择的过程是关键,并且没有它,设计将变得不可能。有趣的是,这个过程可能导致设计的失败。
- 设计需要美学。人机接口领域的研究人员经常把他们的研究限制在和计算机工作的内部方面。然而,作为人类,我们对世界的体验还包括了美学和情感。当人们体验一个软件时,他们体验到的是漂亮、满足和乐趣(或者相反)。然而,美学并不仅仅是视觉表现(尽管其很重要)。Smith 和 Tabor 认为:在程序中存在一种乐趣,它是完全一致的,优雅的;其中,声音和视觉完全是一个整体,或者所选择的表示完全符合用户所想的^[240]。
- 设计是信息的传递。被设计的产品传递着内容,并且经常是多个层面的。有技巧的设计人员知道如何管理这些含义的不同层面。此外,一个激活的产品(例如:咖啡销售机或一个软件程序)传递给用户的是它的使用。这些消息同时被物理机制和便利性所约束。对于一个更广泛的设计语言,其中传递的多种含义包括了功能的、认知的、隐含的和美学的信息^[242]。人不是孤立的接近产品。每个对象表现出来的是期望的上下文,它同时产生于先前对象和经验的历史,以及外围的环境,包括:物理的、社会的和历史上下文^[243]。任何使用计算机的个人都进入

到一个与机器和软件交流的关系中。用户接口可以被看成交流的场所,甚至,绝大部分用户友好的软件可以对用户屏蔽一些隐藏的错误信息。

- 设计是一个社会性行为。关注于单个设计人员的活动,我们可能陷入这样一个错误,即假设一个设计的总的质量主要是创造性个体的活动和质量的结果。然而,设计人员总是在一个更大的设置中操作,它同时受与之交互的其他人员的约束和辅助。
- 设计是创造性的。一个设计在许多约束中为实现一系列目标而被独特的构造。在解决新问题中,并且面临一组新的约束时,设计人员产生新的想法,新的策略和新的解决方案。这些影响和因素的汇合使得设计过程具有了创造性。每个设计都是独特的,因为每个设计试图创建一个可适应的产品。
- 设计是有情感的(在认知和情感中均衡)。由于设计的诞生是一个在设计人员和设计之间的艰苦的探索过程,因此,设计人员经常对所构造的产品赋予了情感是一点都不令人惊讶的。作为一个结果,设计充满了设计人员的精力和灵感。这个情绪和精神可能同时具有建设性和破坏性。它可能激励和加强一个设计的增长,或者可能阻碍它。这个情绪驱动了设计人员去保护他的设计,通常使得设计人员看不到建设性的批评和变化。
- 设计需要一个终结。Frank Kermode 认为艺术最吸引人的地方在于它提供了“一个结尾场景”^[244]。在设计中,总体肯定大于其部分的和。没有这个终结,最终的产品是不完整的,并且会导致设计不稳定、无法设置且无用。
- 设计是一个不断前进的会话。设计的进行过程在两个层次上反复:一个是设计人员的反复,作为一部分当前工作的开发;另一个是由一个团队进行的反复,作为后续的版本。设计是一个内在复杂的过程,设计人员做出的每个选择有着预料的和不可预料的后果。设计不仅仅是小心计划和执行的过程,更多的是一个会话的过程。在这个会话中,会话参与者可能产生不期望的中断和贡献。这个会话通常发生在多个层次上:理论和实践之间;约束和平衡之间;设计人员和材料之间;设计人员和用户之间。随着和材料的会话,设计的理解进入到了设计过程的核心。一个会话意味着交互作用是双向的、无限制的。

15.1.2 软件构架设计

设计经常会触及到构架设计、高层设计、概要设计和详细设计方面的内容。对于一个简单的系统,我们很少去区分这些设计之间的差别。而对于一个复杂的系统,不同的设计层次体现出了一个由抽象到具体的思维过程,在这个过程中,构架设计代表了系统的最早设计判定。构架设计这个领域的最早出现可以追溯到 20 世纪 60 年代^[245],并在后面二三十年中得到了飞速发展^{[246][247][248][249]}。在 Rational 的 RUP 过程中,构架设计处在了极其核心的位置^[145]。一般来说,对于大型复杂的软件项目来说,进行构架设计主要有三个方面的原因^[250]:

- 在各风险承担者之间进行交流。软件构架表示了一个系统的公共的高层抽象,绝大多数风险承担者(如果不是全部的话)可以使用这个基础来建立共同的理解、

形成一致的意见、并且彼此进行交流。

- 早期的设计判定。一个系统的软件构架是最早的产品，它使得可以在众多彼此竞争的关注问题上进行优先级分析，并且展示了系统涉及的质量属性。包括均衡性能和安全性、可维护性和可靠性以及当前开发工作量成本和将来开发工作量成本。
- 一个系统的可传递的抽象。软件构架建立了一个关于系统如何被构建以及系统组件如何一起工作的相对较小的、可以理解的模型；这个模型穿越于整个系统；尤其是，它可以被应用到其他有类似需求的系统，并且改进大规模的复用。

那么，什么是软件构架？SEI收集了大约有60多种关于构架的定义^[253]。Bass, L. 认为“一个程序或计算系统的软件构架是系统的结构，它包含了软件的组件，这些组件的外部可视属性，以及它们之间的关系”^[249]。

RUP中使用了David Garlan和Mary Shaw的定义：软件构架是有关如下问题的设计层次：“在计算的算法和数据结构之外，设计并确定系统整体结构成为了新的问题。结构问题包括总体组织结构和全局控制结构；通信、同步和数据访问的协议；设计元素的功能分配；物理分布；设计元素的组成；指标与性能；备选设计的选择。”^[252]

注意，这里涉及到几个关键点：

- 构架是一个系统或多个系统的一个抽象。它使用有外部可视属性和关系的抽象组件来表示系统；
- 由于构架是抽象的，它抑制了纯粹的局部信息，因此私有组件的细节不在构架范围内；
- 系统是由许多结构组成的（通常称为视图 View）。因此，没有所谓的某个系统的一个构架，并且没有哪个单个的视图可以适当地表示任何东西。此外，视图集不是固定的。构架应当被描述成一组视图，这组视图支持构架分析和交流的需要。

1. 软件构架视图

正如定义所指出的，软件的构架需要通过一组视图表示出来。常见的视图一般包括如下几种^{[255][256]}。

- 功能/逻辑视图 (Functional/Logical View)：功能视图是系统功能和它们关系的一个抽象。功能视图的组件包括：功能、关键系统抽象和域元素。在视图中，组件间的关系使用依赖关系和数据流表示。参考图 15-1。该视图的用户一般包括领域工程师、产品线设计人员和最终用户；

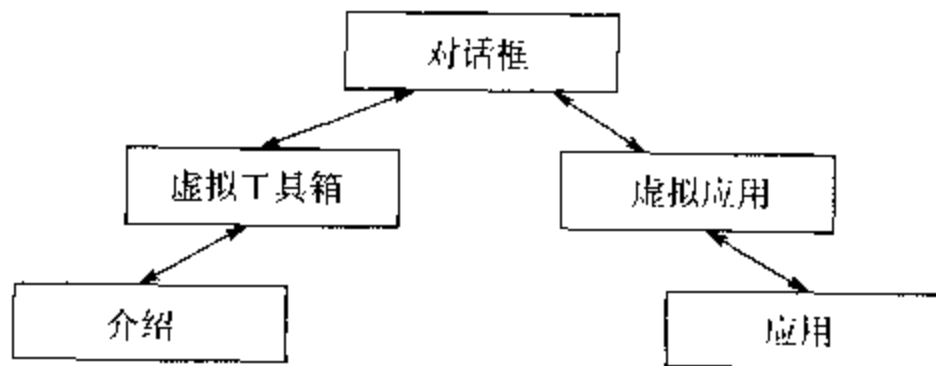


图 15-1 功能视图例子

- 代码视图(Code View): 该视图是从程序员的视角出发的, 因此, 该视图的组件一般是类、对象、过程和函数, 以及它们的抽象/或组合而成的子系统、层和模块。组件间的关系一般是调用关系或包含关系。参考图 15-2。该视图的用户包括程序员、设计人员和重用人员(一般也是程序员或设计人员);

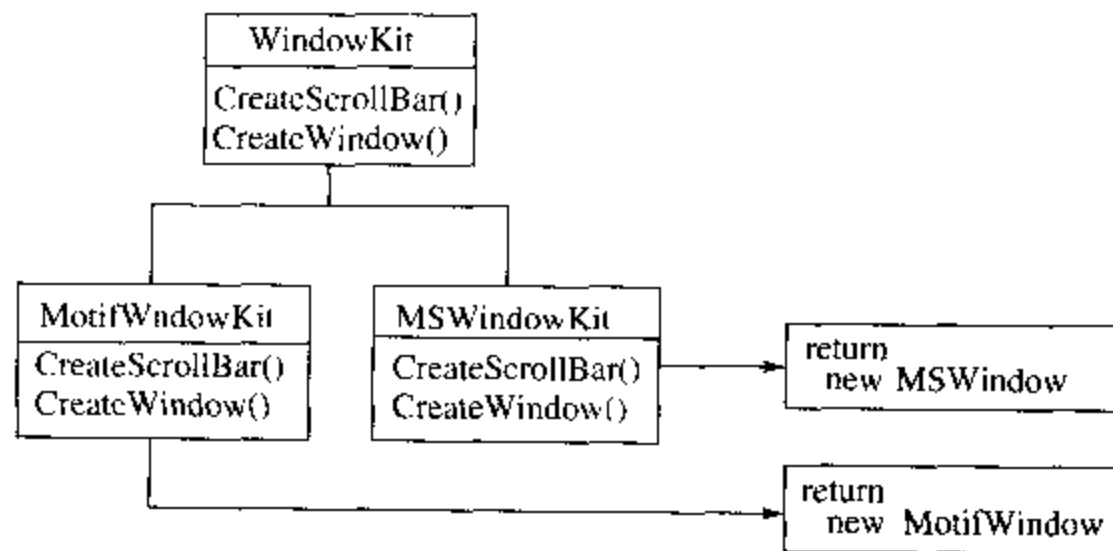


图 15-2 代码视图例子

- 开发/结构视图(Development/Structural View): 开发视图也是开发人员看的视图, 但不同于代码视图, 它是一个源代码结构的视图, 作为一个仓库, 由不同的用户(程序员或维护人员)创建、修改和管理。这个视图的组件一般是文件和目录。其关系一般是包含关系。参考图 15-3。该视图的用户包括程序员、维护人员、经理、配置经理;

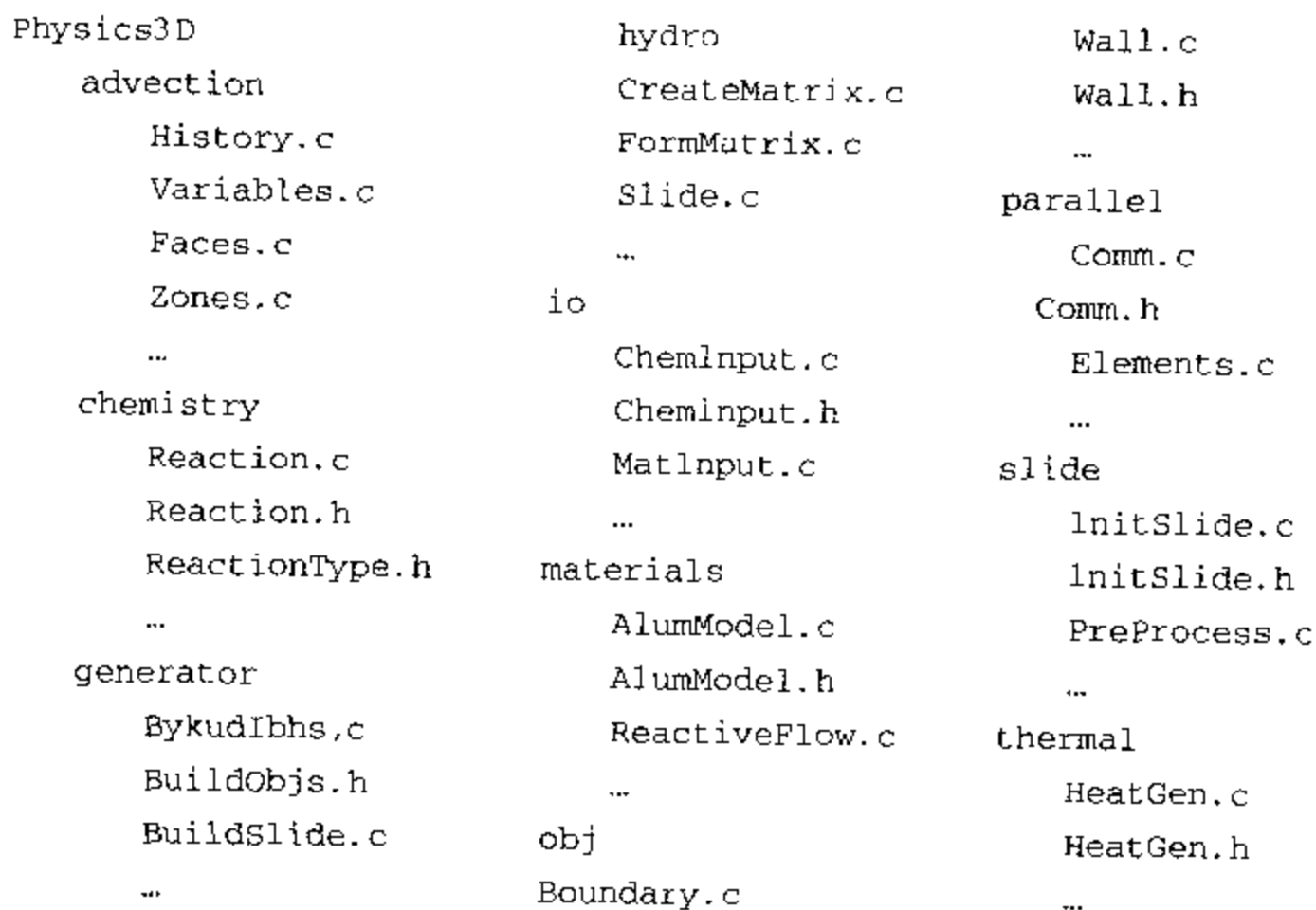


图 15-3 开发视图例子

- 并发/进程/线程视图(Concurrency/Process/Thread View): 当设置一个复杂的系统的时候, 它一般被包含到一系列的进程或线程中, 这些进程和线程分布在一些可

计算的资源上。并发视图是一个必须的步骤用于论证什么进程或线程将会被创建以及它们如何通信和共享资源。这个视图的组件是进程和线程，它们之间通过共享资源上的数据流、事件和同步来接口，参考图 15-4。该视图的用户包括那些关心系统配置，关心系统可用性和性能的人员，以及集成人员和测试人员；

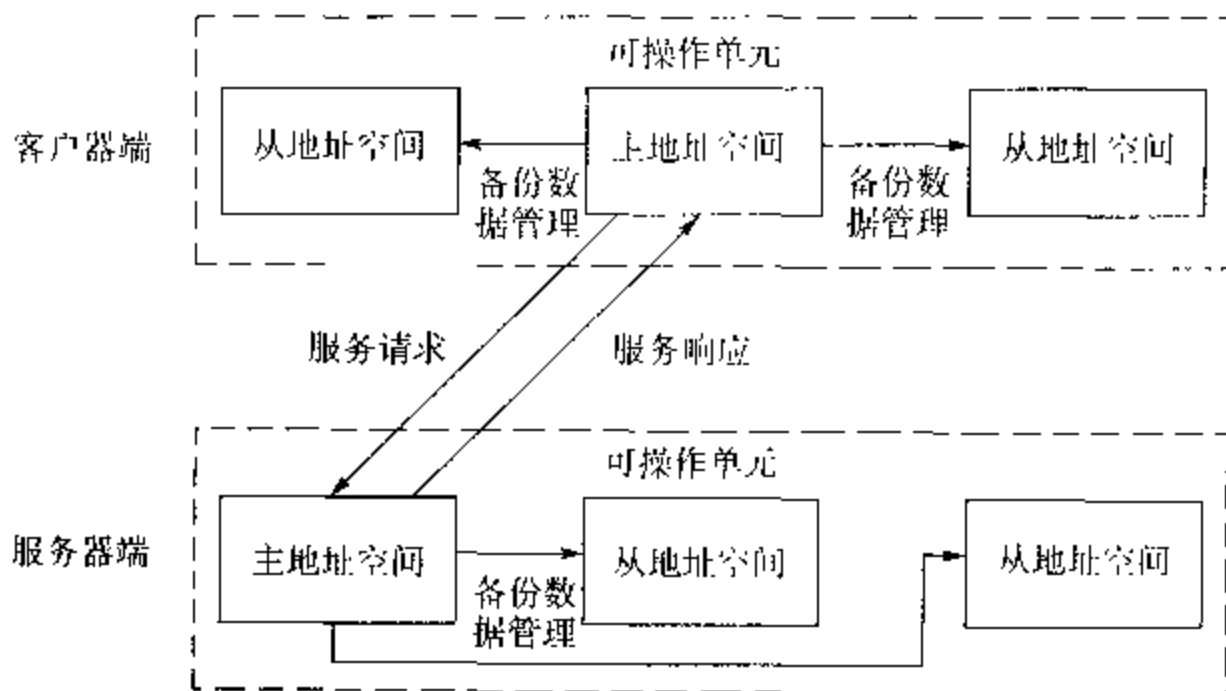


图 15-4 并发视图例子

- 物理/发布视图 (Physical/Deployment View)：物理视图描述了系统是如何分布在硬件资源上的。对于小的、价值不高的系统，这个描述同样是微不足道的：一台计算机，所有进程都在该计算机上运行。但是对于大型复杂的系统，可能存在无数传感器、制动器、存储设备、网络 and 计算机。这个视图的组件包括：CPU、传感器、制动器和存储器。它们之间的连接一般是网络或其他通信设备（例如卫星或总线），参考图 15-5。该视图的用户是硬件或系统工程师。

Philippe Kruchten 把这 5 类视图合称为 4 + 1 视图^[254]，RUP 中使用了这个分类，由于使用了基于 UML 的方法，视图的名称上略有不同，具体如下所述。

- 用例视图 (Use-Case View)：包括用例和场景，这些用例和场景包括在构架方面具有重要意义的行为、类或技术风险。它是用例模型的子集；
- 逻辑视图 (Logical View)：包括最重要的设计类、从这些设计类到包和子系统的组织形式，以及从这些包和子系统到层的组织形式。它还包括一些用例实现。它是设计模型的子集；
- 实施视图 (Implementation View)：包括实施模型及其从模块到包和层的组织形式的概览。同时还描述了将逻辑视图中的包和类向实施视图中的包和模块分配的情况。它是实施模型的子集；
- 进程视图 (Process View)：包括所涉及任务（进程和线程）的描述，它们的交互和配置，以及将设计对象和类向任务的分配情况。只有在系统具有很高程度的并行时，才需要该视图。在 Rational Unified Process 中，它是设计模型的子集；
- 发布视图 (Deployment View)：包括对最典型的平台配置的各种物理节点的描述以及将任务（来自进程视图）向物理节点分配的情况。只有在分布式系统中才需

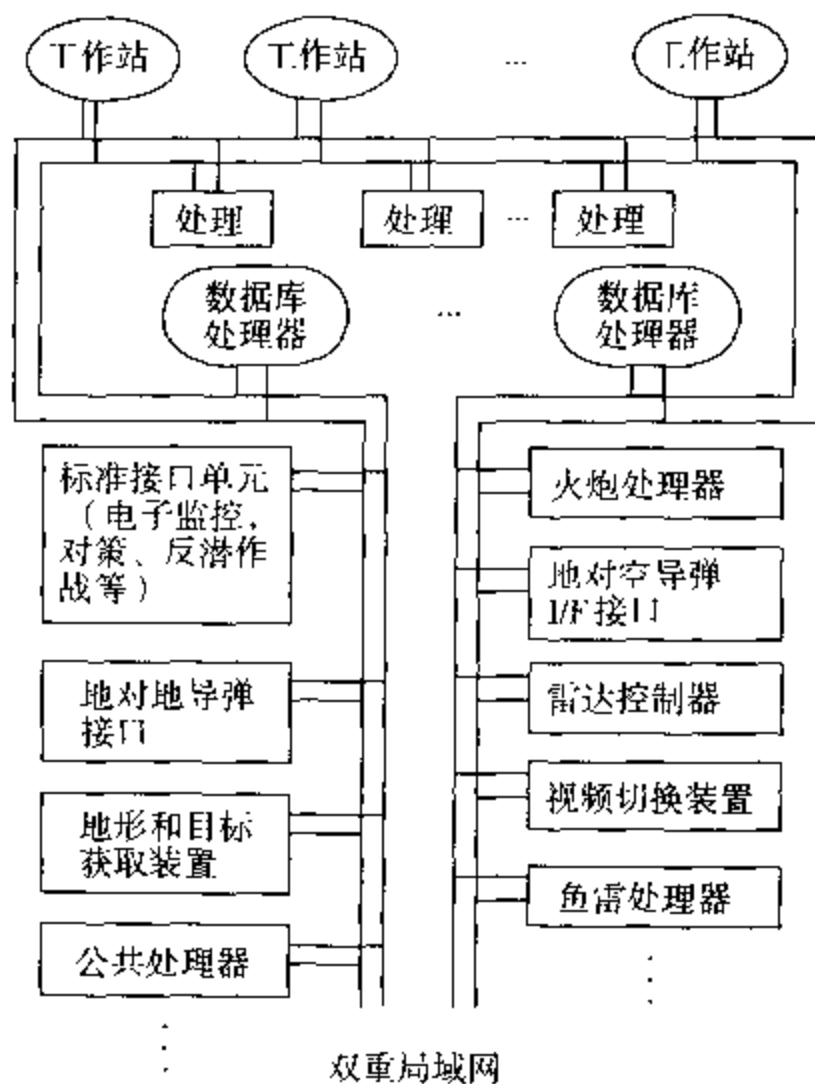


图 15-5 物理视图例子

要该视图。它是配置模型的一个子集。

2. 场景

除了软件构架视图外，构架师使用场景 (Scenarios) 把所有视图联接在一起来表示构架^[254]。场景是一个系统的使用或变更的一个简单描述。它们通常可以分成如下两个子类。

- 使用实例 (Use Case)：责任的顺序；
- 变更实例 (Change Case)：对系统建议变更的描述。

场景被用于：

- 通过构架师、设计人员和程序员“走读”构架来理解和验证构架；
- 把构架信息传递给那些在构架创建过程中不担任主要角色的人员；
- 把视图联系在一起，就如我们所看到的，视图很少孤立地出现。一个视图一般会映射到另一个视图。场景帮助了这个映射过程；
- 通过寻找构架难以满足的地方或者应用于满足场景的地方来理解构架的限制。

15.1.3 概要设计和详细设计

软件设计是一个把软件需求变换成软件表示的过程。最初这种表示只是描绘出软件的总的框架，然后进一步细化，在此框架中填入细节，把它加工成在程序细节上非常接近于源程序的软件表示。

从工程管理的角度上来看, 软件设计分为两个步骤。首先是概要设计, 将软件需求转换为数据结构和软件的系统结构。然后是详细设计, 即过程设计。通过对结构表示进行细化, 得到软件的详细的数据结构和算法。

构架设计是概要设计的最顶层设计, 对于一个复杂的系统往往还会把概要设计分为0层设计、1层设计和2层设计。在构架设计中一般不会涉及到数据结构的设计以及系统结构中外部不可视的模块。然而对于概要设计来说, 最终需要定义出这些内容。

从结构化设计的角度来看, 概要设计需要具体完成的工作有以下几项。

- 制定规范

规范的制定是为了让开发组在设计时遵守共同的标准, 以便协调各成员的工作, 它包括:

(1) 阅读和理解软件需求说明书, 在给定的预算范围内和技术现状下, 确认用户的要求能否实现, 从而确定设计的目标, 以及它们的优先顺序;

(2) 根据目标确定最合适的设计方法;

(3) 规定设计文档的编制标准, 包括文档体系、用纸样式、记录的详细程度、图形的画法等;

(4) 规定编码的信息形式, 与硬件/操作系统的接口规约、命名规则等。

- 软件系统结构的总体设计

该部分工作包括以下内容:

(1) 采用某种设计方法, 将一个复杂的系统按功能划分成模块的层次结构;

(2) 确定每个模块的功能, 建立与已确定的软件需求的对应关系;

(3) 确定模块间的调用关系;

(4) 确定模块间的接口, 即模块间传递的信息。设计接口的信息结构;

(5) 评估模块划分的质量及导出模块结构的规则。

- 处理方式设计

(1) 确定为实现软件系统的功能需求所必须的算法, 评估算法的性能;

(2) 确定为满足软件系统的性能需求所必须的算法和模块间的控制方式。性能主要是指周转时间、响应时间、吞吐量、精度;

(3) 确定外部信号的接受发送形式。

- 数据结构设计

确定软件涉及的文件系统的结构以及数据库的模式、子模式, 进行数据完整性和安全性的设计。它包括:

(1) 确定输入、输出文件的详细的数据结构;

(2) 结合算法设计, 确定算法所必须的逻辑数据结构及操作;

(3) 确定对逻辑数据结构所必须的那些操作的程序模块。限制和确定各个数据设计决策的影响范围;

(4) 若需要与操作系统或调度程序接口所必需的控制表等数据时, 确定其详细的数据结构和使用规则;

(5) 数据的保护性设计, 包括防卫性设计、一致性设计和冗余性设计。

- 可靠性设计

- 编写概要设计文档
- 评审概要设计

对于详细设计来说，需要完成的工作有：

- 确定软件各个组成部分内的算法以及各部分的内部数据组织
- 选定某种过程的表达形式来描述各种算法
- 进行详细设计评审

15.2 设计的评审

类似于需求，我们可以通过同行评审来对设计进行验证，有关同行评审的概念请参考第16章的内容。

15.2.1 设计查检表

查检表是评审时使用的一个重要工具。这里分别给出概要设计和详细设计的查检表（见表15-1和表15-2）。

表 15-1 概要设计查检表

检查要素	检查内容
概述	是否准确且充分阐述了设计系统在项目软件中的地位和作用，其与同等、上级系统的关系是否描述
系统描述和可追溯性	需求规格概述是否与需求规格说明书一致？应具体到需求配置项的引用
	是否每一部分的设计都可以追溯到需求说明书，接口规格说明书或其他产品文档？应具体到配置项的引用
	是否对需求分析中不完整、易变动、潜在的需求进行了相应的设计分析
	模块的规格是否和软件需求文档中的功能需求和软件接口规格要求保持一致
	设计和算法是否能满足模块的所有需求
	是否阐述了设计中的风险和对风险的评估
总体设计	设计目标是否明确清晰地进行了定义
	是否阐述了设计所依赖的运行环境，以及和需求中运行环境的一致性是否满足
	是否全面准确地解释了设计中使用到的一些基本概念
	设计中的逻辑是否正确和完备
	是否全面考虑了各种设计限制
	是否有不同的设计方案的比较？是否有选择方案的结论？是否清楚阐述了方案选择的理由
	是否合理地划分了模块并阐述了模块间的关系
	系统结构和处理流程能否正确地实现全部的功能需求

续表

检查要素	检查内容
接口设计	用户接口设计是否正确且全面? 是否有单独的用户界面设计文档
	是否有硬件接口设计? 硬件接口设计是否正确且全面
	是否有软件接口设计? 接口设计是否正确且全面
	是否有通信接口设计? 通信接口设计是否正确且全面
	内部接口设计是否正确且全面
	是否描述了接口的功能特征
	接口是否便于查错
	接口相互之间、接口和其他模块、接口和需求说明书及接口规格书是否保持一致
	是否所有的接口都需要类型、数量、质量的信息
属性设计	是否有可靠性的设计, 设计是否具体、合理、有效
	是否有安全性的设计, 设计是否具体、合理、有效
	是否有可维护性的设计, 设计是否具体、合理、有效
	是否有可移植性的设计, 设计是否具体、合理、有效
	是否有可测试性的设计, 设计是否具体、合理、有效
	是否明确规定了测试信息的输出格式
数据结构	是否准确定义了主要的常量
	全局变量的定义是否准确
	定义的全局变量的必要性是否充分
	主要的数据结构是否都有定义
	是否说明了数据结构存储要求及一致性约束条件
	是否对所有数据成员、参数、对象进行了描述
	是否所有需要的数据结构都进行了定义, 或者定义了不需要的数据结构
	是否所有的数据成员都进行了足够详细的描述? 数据成员的有效值区间是否定义
	共享和存储数据的使用是否描述清楚
运行设计	对系统运行时的顺序、控制、过程及时间的说明是否全面、准确
出错处理	是否列出了主要的错误类别, 每一错误类别是否都有对应的出错处理
	设计是否考虑了检错和恢复措施? (例如: 输入检查)
	出错处理是否正确、合理
运行环境	硬件平台、工具的选择是否合理
	软件平台、工具的选择是否合理

续表

检查要素	检查内容
调试方法	调试方法是否完备且可实现
	测试用例是否包含输入的合理等价和不合理等价值
	测试用例是否有合法边缘值和非法边缘值
	是否有针对函数入口、出口参数的调试方法和信息
	是否有针对运行流程的调试方法和信息
清晰性	程序结构, 包括数据流、控制流和接口的描述是否清楚
一致性	程序、模块、函数、数据成员的名称是否保持一致
	设计是否反映了真正的操作环境、硬件环境、软件环境
	对系统设计的多种可能的描述之间是否保持一致? (例如: 静态结构的描述和动态描述)
	设计在计划、预算、技术上是否可行
详细程度	是否估计了每个子模块的规模(代码的行数)? 是否可信
	程序执行过程中的关键路径是否都被标名和经过分析
	是否考虑了足够数量及代表性的系统状态
	详细程度是否足够进行下一步的详细设计
可维护性	是否模块化设计
	模块是否为高内聚、低耦合
性能	是否进行了性能分析? 是否有存在论证过程的性能数据和规格
	是否描述了所有的性能参数? (例如: 实时性能约束、存储空间、速度要求、磁盘 I/O 空间)

表 15-2 详细设计查检表

检查要素	检查内容
清晰性	是否所有的单元和进程的设计目的都已文档化
	单元设计, 包括数据流、控制流、接口描述是否清楚
	单元的整体功能是否描述清楚
完整性	是否提供了所有程序单元的规格
	是否描述了所采用的设计标准
	是否确定了单元应用的算法? (例如: PDL)
	是否列出了单元的所有调用
	是否记录了设计继承的历史和已知的风险
规范性	文档是否遵从了公司的标准
	单元设计是否使用了要求的方法和工具

续表

检查要素	检查内容
一致性	在单元和单元的接口中数据成员的名称是否保持一致
	所有接口之间, 接口和接口规格书之间是否保持一致
	详细设计和概要设计文档是否能够完全描述“正在构建”的系统
正确性	是否有逻辑错误
	需要使用常量名称的地方是否有错误
	是否所有的条件都被处理? ($>$, $=$, <0 , switch case)
	分支所处的状态是否正确? (逻辑是否搞反?)
数据	是否所有声明的数据块都已经使用
	定位于单元的数据结构是否已经描述
	如果有对共享数据、文件的修改, 对数据的访问是否按照正确的共享协议进行? (例如: 通过信号灯同步进程)
	是否所有的逻辑单元、事件标记、同步标记都已经定义和初始化
	是否所有的变量、指针、常量都已经定义并初始化
功能性	设计是否使用了指定的算法
	设计是否能够满足需求和目的
接口	参数表是否在数量、类型和顺序上保持一致
	是否所有的输入输出都已经正确定义并检查过
	所传递参数的顺序是否描述清楚
	参数传递的机制是否确定
	通过接口传递的常量和变量是否与单元设计的相同? (例如, 函数中定义的常量不能在所调用的子过程中被修改)
	传入、传出函数的参数, 控制标记是否都已经描述清楚
	是否以度量单位描述了参数的值区间、准确性和精度
	过程对共享数据的理解是否一致
详细程度	代码和文档间的展开率是否小于 10:1
	对模块的所有需求都已经定义
	详细程度是否足够开发和维护代码
可维护性	单元是否是高内聚和低外部耦合? (例如: 单元的改变不会在内部出现不可预见的影响, 同时对其他单元的影响最小?)
	是否这种设计是复杂度最小的设计
	开始部分的描述是否符合公司的要求? (如: 目的、作者、环境、非标准特性、开发历史、输入输出参数、使用的文件、数据结构、引用此单元的其他单元、注释)

续表

检查要素	检查内容
性能	进程是否有时间窗
	是否所有的时间和空间的限制都已明确
可靠性	初始化时是否使用了默认值, 是否正确
	访问内存时是否进行了边界检查, 以保证地址正确? (队列、数据结构、指针等)
	对输入、输出、接口和结果是否进行了错误检查
	对所有错误情况都安排了有意义的消息反馈
	特殊情况下的返回码是否和文档中定义的全局返回码一致
	是否考虑了异常情况
可测性	是否每个单元都可以被测试、演示、分析或者检视, 以确认满足需求
	设计中是否包括辅助测试的检查点? (例如: 条件编译代码、断言等)
	是否所有的逻辑都是可测的
	是否描述了本单元的测试驱动模块、测试用例集、测试结果
可追溯性	是否每一部分的设计都可以追溯到需求
	是否每一个设计决策都可以追溯到效益分析
	是否所有的设计决策都可以追溯到成本/效益分析
	是否描述了每个单元的详细需求
	单元需求是否能够追溯到软件规格文档(SSD-1)? 软件规格文档是否能够跟踪到单元需求
	是否有到代码的引用或者包括代码本身

15.2.2 构架设计评审方法

对软件构架进行评价主要是为了在产品被开发之前尽早发现质量问题。传统的评审方法可以被应用到构架设计评审中。由于在构架设计中使用了场景的概念, 因此在这个基础上产生了一些特定的构架分析方法, 包括诸如: SAAM(Software Architecture Analysis Method)^{[249][257][260][262]}, ASAAM(Aspectual Software Architecture Analysis Method)^[258], ATAM(Architecture Tradeoff Analysis Method)^[259]等。在这众多模型中, SAAM 是基本的软件构架分析方法, 其他模型都是以这个为基础的, 其中 ASAAM 是 SAAM 的一个扩展和优化, 它除了包含了 SAAM 的基本方法和步骤之外还提出了一些补充技术。ATAM 的一个最大的区别在于它除了揭示构架是否满足特定的质量目标之外, 还提供了质量目标如何与其他目标进行平衡的一个透视, 它们有着最深远的影响, 并且在系统一旦被实现之后是最难以变更的。在本节, 将重点介绍 SAAM 方法, 并且简单地介绍 ASAAM 和 ATAM 这两种方法。

1. 软件构架分析方法

软件构架分析方法(SAAM)是基于软件构架的评价对系统级软件质量属性做出预计。SAAM使用了来自于计算机系统中不同角色的场景,主要是为了评价竞争的构架。该方法是SEI建议的方法^[256]。SAAM评审方法是一个现代的评审技术,并且正变得越来越流行。

- 关于输入

很明显,构架的规格和描述是需要的。由于构架可以从很多细节层次以及许多视图来进行描述,因此,哪一类规格和描述需要评价呢?答案是,凡是用于解答评价技术所关注的问题的任何规格和描述都是评审的输入。例如,如果是为了评价性能和并行性方面的问题,那么构架的任务以及通信结构可能需要被作为输入;如果需要评价可修改性,那么构架的模块分解和工作分解将被需要。

- 关于输出

输出依赖于选择的方法。SAAM产生一个相关的等级划分。即,如果两个或两个以上的候选构架被用于比较哪个更满足需求,那么SAAM将对所有候选构架产生一个等级划分。如果只有一个构架被用于评价,那么SAAM将指出构架哪些地方没有能够满足质量需求,并且在某些情况下指出可替换的更好的设计。这个结果是粗糙且广泛的,它反映了在一个非常粗的粒度下进行构架分析的内在变化。

- 使用场景分析质量

人们希望通过分析软件构架来评价其质量属性,类似可维护性、安全性、性能、可靠性等。尽管IEEE标准已经定义了很多如何评价这些质量属性的标准。然而,绝大多数的质量属性太复杂和太抽象以致无法使用一个简单的指标来衡量。例如一个系统仅仅可以通过变更数据文件来改变用户接口背景颜色,但是相同的系统需要手工变动几十处程序来适应新的数据存储设计。你认为该系统是可修改的还是不可修改的呢?

假如一个系统的用户接口被仔细地思考,以便一个新的用户只需要简单的培训就可以操作系统,但是有经验的用户发现它是如此冗长乏味,以至不喜欢使用。你认为这个系统是可用性还好还是不好?

当然,关键点在于质量属性不是孤立存在的,而需要在一个上下文之内才有意义。一个系统不可修改要根据特定的变更类型,是否安全要根据特定的威胁,是否可用要根据特定的用户群,是否有效要根据特定资源的使用等。因此说“这个系统是高可维护性的”是完全没有可操作意义的。

对质量属性的基于上下文评价的概念使我们采纳了场景(Scenarios)作为评价质量的手段。SAAM就是基于场景的构架分析方法。它提供了一个方法用于确定一个特定的构架设计根据使用在该构架上的一组场景要求来判定构架有多好,其中,场景是包含系统的使用或修改的一系列步骤。这样,很容易想象一组场景来测试我们所关心的可修改性、安全性、性能等质量属性。

实际上,一个特定的场景可以作为某类所有场景的代表。这类场景包含了用于被评价系统对应于一个相同方法的所有场景。例如,场景“把所有窗口背景色改成蓝色”在绝大多数系统中是完全等同于场景“变更所有窗口的边界装饰”。SAAM度量有多少场景对相同构架组件产生变更;这称为聚类。作为一个结果,判断需要根据聚类场景在相同的主题

上是表现了一些变异还是根本完全不同。如果是第一种情况,那么聚类是好事;否则是坏事。换句话说,如果一组场景是类似的,并且它们都影响了构架中的相同组件,那么它们是好的,因为这意味着系统的功能已经根据可能的修改任务被适当地模块化了。另一方面,如果一组类似的场景影响许多不同的组件,那么就很糟糕了。

为了帮助创建和组织场景,SAAM使用了与系统相关的角色的概念。角色的例子包括:负责升级软件的人员,负责管理系统数据库的人员,负责批准系统新需求的人员等。

• 评价步骤

下面我们将列出SAAM评价的步骤,并不是所有SAAM需要执行所有步骤的,项目可以从这些步骤中的任何一个子集中获得好处。

步骤1:开发场景

开发系统随时间变化可能产生的场景,这些场景用图表示了各种系统必须支持的活动和各种系统被预期可能的修改。这样场景可以代表与不同角色相关的任务。

步骤2:描述候选构架

构架或候选构架应当使用系统化的构架描述语言进行描述,这个描述语言应当能够被参与分析的所有人员很好地理解,类似UML等。这些构架描述需要指出系统的计算和数据组件,以及所有相关的连接。SAAM评价已经趋向于使用非常简化的构架原始物。一个典型的代表是要能够区分数据连接(在组件间传递信息)和控制连接(一个组件控制另一个组件执行它的功能)。这个简单的词法提供了构架的一个合理的静态表示。伴随着这个构架静态表示的是一个关于系统行为是如何随时间变化的描述,或者一个更为动态的构架表示。这可以采用整个行为的自然语言规格描述格式或者一些其他更正式的和结构化的规格。

步骤3:对场景分类

在场景类型之间存在着一个重要的区别。想象一下,场景是一个系统一些预计或期望使用的简单描述。它可能是系统直接支持的那种场景,意味着预计的使用对系统执行起来是不需要修改的。这一般可以通过证明已有构架在执行场景(更像是在构架级的一个系统模拟)时表现如何来确定。如果一个场景不是直接支持的,这意味着必须对系统做一些修改才能表示构架。这可能会改变一个组件或多个组件如何执行被分配的活动,一个组件如何额外执行一些活动,已存在组件之间如何连接,或者包括上面所有这些的组合。我们把第一类场景称为直接场景,第二类场景称为非直接场景。对于一个构架来说,一个特定的场景只能是直接的或非直接的。在使用SAAM时,无论场景是直接的还是非直接的,经常的分析之前是非常引人注目并且经常对客户是未知的。很明显,对于引人注目的场景是直接支持的构架比那些需要修改才能支持相同场景的构架更有能力,并且在SAAM中获得更高的分数。

步骤4:执行场景评价

对于每个非直接场景,列出构架为支持场景所必须的变更。对构架的一个修改意味着要么一个新的组件,要么一个新的连接被引入,或者一个已有的组件或连接需要改变它的规格。这个阶段结束时,应当有一个总结表列出所有的场景(包括直接的和非直接的)。对每个非直接的场景,这些场景对构架产生的影响或变更应当被描述。其困难的权重也应当在这个阶段被完成。通常,这个权重粒度是很粗的,是基于对构架的理解。在比较可替

换的候选构架时，一个总结表格特别有用，因为它提供一个简单的方法来确定哪个构架更好地支持了场景的集合。

步骤5：揭示场景交互

当两个或多个非直接任务场景需要改变一个系统的一个组件的时候，它们被认为是与那个组件有交互。场景交互是重要的，因为它揭露了产品设计中的功能分配。语法上不相关场景的交互明确地显示了哪个系统模块计算了语法上不相关的功能。高的场景交互区域揭示了在一个系统组件中，其内容的独立性比较糟糕。这样，场景交互的区域表示是设计人员随后需要重点关注的地方。场景交互的数量与诸如结构复杂性、耦合性和凝聚性的度量相关。因此，它可能和最终产品的缺陷数量非常相关。

步骤6：总体评价

最后，根据相关重要性，给每个场景和场景交互分配一个权重，并且使用这个权重去确定一个总的级别。通过了解哪个场景更重要，联结就可以被打开。分配权重是一个主观的过程，涉及到系统中所有的风险承担者。除了提供一个简单的构架度规外，SAAM 还提供一套小的度量规则。有了这组小型的度量，SAAM 就可以根据每个场景用于比较竞争的构架。剩下的就是由 SAAM 的用户来决定哪个场景对他们最重要。此外，执行 SAAM 分析的过程还被用于用户获得竞争构架的一个高度了解。

构思场景和表示构架是彼此依赖的步骤。决定一个构架的适当的粒度级别依赖于你想要评价的场景的种类。确定一个合理的场景集合也依赖于你期望系统能够执行的活动的种类，并且它还受构架影响。图 15-6 是 SAAM 评价步骤的一个图示。

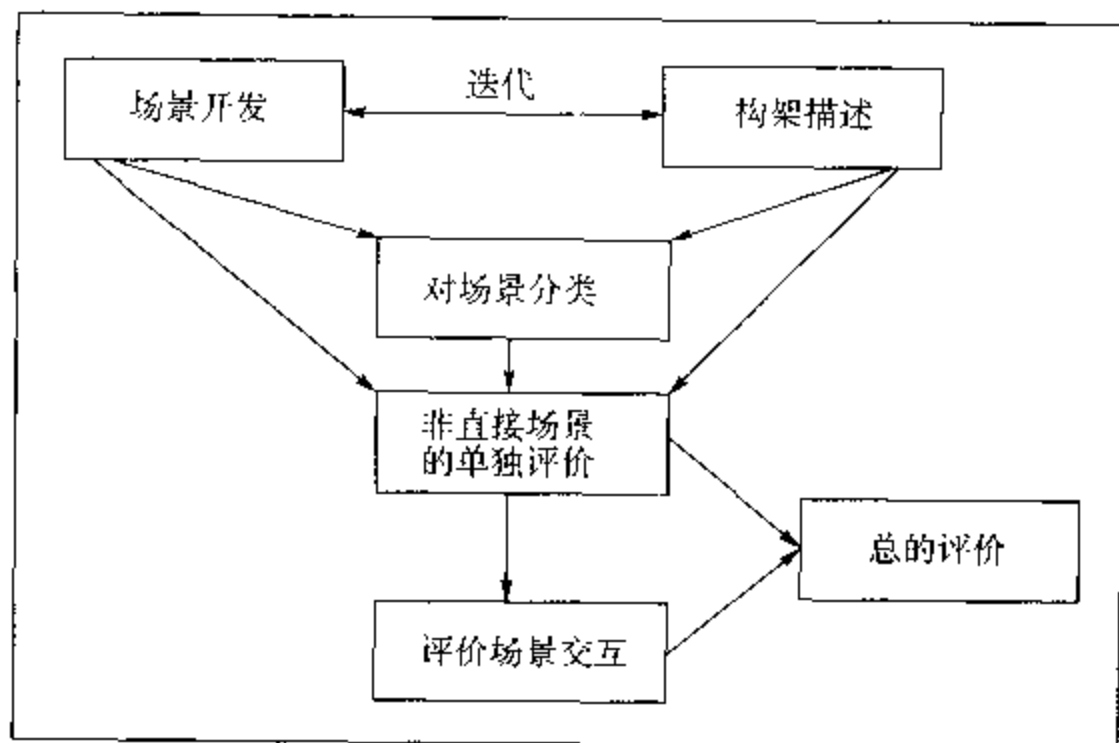


图 15-6 SAAM 分析的活动及依赖关系

2. 表面的软件构架分析方法

表面的软件构架分析方法 (ASAAM) 是在软件构架设计中用于评价软件构架表面的一种方法。该方法不同于当前其他的一些软件构架分析方法，它在构架设计层关注横穿于多个构架组件之间的属性。ASAAM 使用了一套启发式的规则来帮助从场景中引出构架的表面和相应的纠缠在一起的构架组件^[258]。

ASAAM 是以 SAAM 为基础, 并且在此基础上进行了扩展和优化。ASAAM 的基本活动可以使用图 15-7 来表示。

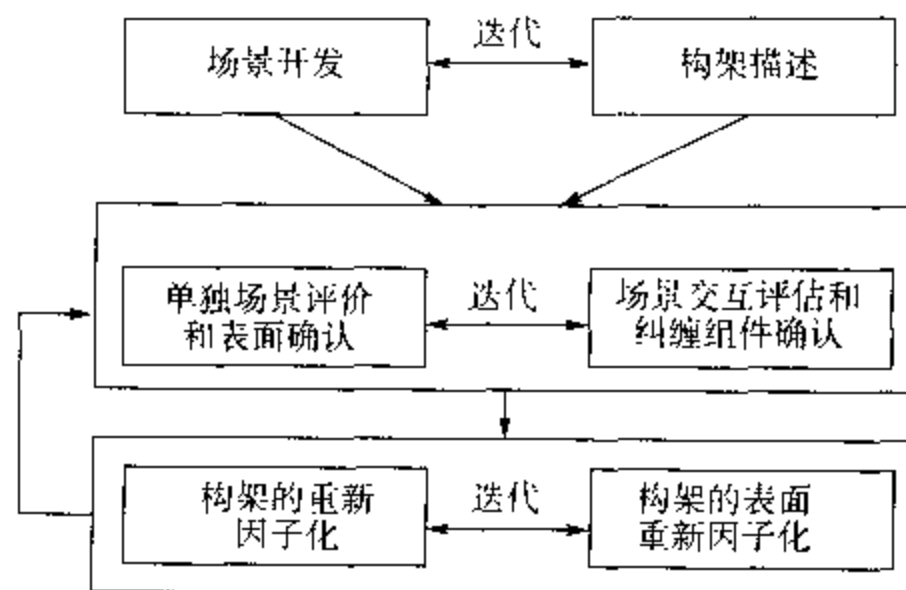


图 15-7 ASAAM 分析的活动和依赖关系

步骤 1: 候选构架开发

根据需要的质量因子和潜在的面提供候选的构架。

步骤 2: 开发场景

类似 SAAM。

步骤 3: 独立的场景评价和表面确认

场景首先被分类成直接和非直接场景。作为 SAAM 的补充, 场景评价还寻找潜在的构架表面。启发式规则的应用导致场景被进一步分类为: 直接场景、间接场景、表面场景和构架表面。表面场景来自于直接或间接场景, 并且代表潜在的表面。通过表面域分析, 用于场景的对应表面被搜索和描述。

步骤 4: 场景交互评估和组件分类

这个步骤的目的是评价构架是否支持对关心的内容进行适当的分离。这包括非贯穿于各构架组件的内容和贯穿于各构架组件的内容, 即表面 (Aspect)。对于每个组件, 包括直接和间接组件, 被分析和分类成内聚组件、纠缠组件或者复合组件或不清楚的组件。

步骤 5: 构架再因子化

根据场景交互评估和组件分类, 建议对构架进行再因子化。再因子化可以通过使用传统的抽象技术来完成, 例如: 设计模式, 或者使用面向表面的技术。最后构架表面和纠缠的组件在构架中被清晰地描述。

3. 构架均衡分析方法

构架均衡分析方法 (ATAM) 的目的是使用质量属性需求来评估构架判定的结果。ATAM 是一种风险确认方法, 可以在一个复杂软件构架内部检测潜在风险区域。这有多个含义:

- ATAM 可以在软件开发生命周期早期进行;
- ATAM 可以迅速执行且成本相对较低;
- ATAM 会根据构架规格的详细级别产生相当的分析; 此外, 它不需要产生一个成

功系统的任何可度量质量属性的详细分析。作为替换, ATAM 主要分析系统质量属性的发展趋势。

ATAM 建立基础的一个关键概念是质量属性特征 (Quality Attribute Characterization)。根据质量属性需求评价一个构架设计必须有一个准确的质量属性特征。例如, 要了解一个构架的可修改性, 需要对如何度量或观察可修改性有个了解, 并且需要了解不同的构架结果是如何影响这些度量的。为了使用大量已经存在在不同质量属性共同体中的知识, 我们创建了性能、可修改性、可获得性、可用性和安全性的质量属性特征。这些特征可以作为准备或执行 ATAM 的起始点。

每个质量属性特征包含 3 个内容: 外部激励、构架判定和反应。“外部激励”是可能引起构架反应或变更的事件。为了分析构架是否遵从质量需求, 那些需求需要使用具体的、可度量或可观察的短语进行描述。这些可度量/可观察的数量在“反应”部分被描述。“构架判定”是那些对构架获得属性反应有直接影响的方面, 如: 组件、连接以及它们的属性。^[259]

ATAM 的步骤可以表述如下。

介绍阶段

步骤 1: 介绍 ATAM

步骤 2: 介绍业务驱动

步骤 3: 介绍构架

调查和分析阶段

步骤 4: 确定构架的方法

步骤 5: 产生质量属性用法树

步骤 6: 分析构架的方法

测试阶段

步骤 7: 头脑风暴和区分场景优先级

步骤 8: 分析构架方法

报告阶段

步骤 9: 介绍结果

15.2.3 软件构架评价最佳工业实践

软件构架评价是产品开发过程中的一个重要步骤, 并且已经越来越得到业界的重视。如何做好这个工作的研究也在不断开展, 并获得了很好的工业实践。我们前面所给出的 SAAM、ASAAM 和 ATAM 就是这些好的工业实践中的几种。业界还存在着其他一些用于构架评价的好的方法, 然而, 该如何来选择呢? 卡耐基梅龙大学在其研究和实践的基础上给出了建议的最佳工业实践, 该实践结合了成本、受益和技术的多方面综合^[261]。

1. 成本和受益

进行一个构架评价的动机有很多, 但是一个最重要的动机是考虑与时间及承诺资源相关的成本。对一个组织来说, 最大的问题是要决定我们投入的成本是否能够获得足够的回

报。因此，首先在这里看一些业界关于这方面的实践报告，相信这对你或你的组织决定是否采纳构架评价活动是非常重要的。

在一些已经进行的学术讨论会过程中，参与者维护了一个记录以帮助确定整个构架评价的成本。例如，在 AT&T 公司，已经执行了大约 300 个以上的完整的构架评价，执行这些构架评价的项目最小也需 700 人天以上的工作量，报告显示其构架评价的成本大约 70 人天。IBM Rational 软件公司已经执行了大约 30 个软件构架评价，对于不小于 500KLOC 代码规模的项目来说，其平均成本大约是 5 万美元。SEI 软件构架分析方法(SAAM)评价人员已经执行了 10 个构架评价，使用该评价方法的项目规模在 5 ~ 100KLOC 之间，并且报告的评价成本是 14 人天。绝大多数参与者还注意到在公司中，由于缺乏构架成熟度，开始一个构架评审还需要花费额外的启动成本，这个成本包括构架评价方法的培训，人员的相互熟悉等各方面的工作。

大多数具有代表性的公司(如 AT&T，西门子，诺基亚，摩托罗拉和洛克希德·马丁等公司)已经建立了一个专门的组织来负责定义和执行构架评价。这样做的一个主要原因是能够提供企业的构架评价成熟度，并且在最大限度上提供构架评价效果，降低构架评价成本。

构架评价的好处可以从以下几个方面进行分析：

- 财务方面的好处

AT&T，每个项目经理都报告了从构架评价中获得了成本的节约。在过去 8 年中，进行完整构架评价的项目报告大约降低了 10% 的项目成本。也就是说，对于一个 700 人天或更大的一个项目来说，至少可以获得 70 人天的成本节约。

几个咨询者的报告说，他们公司通过构架评价避免了几百万美元的购买费。其他一些类似报告包括：在一个电子投资交换系统中，早期构架评价分析发现系统每晚只能支持 50 亿美元的峰值交换能力，而这只有期望能力的一半；在一个零售售货系统中，构架评价揭示了可能存在一个峰值订货性能问题，一个主要的业务失败被避免了；在一个修正控制系统中，构架评价发现了系统的许多可移植性和可修改性限制，因此系统被建议进行了重新设计。

给出不进行构架评审的情况也是有用的。一个参与者描述了重写一个客户记账系统是如何被估计需要两年的。几年以后，这个系统已经被重复实现了三次。性能目标从未被满足，尽管最后一个版本所使用的 CPU 能力比最初原型版本所使用的 CPU 能力提高了 10 倍，该项目在花费了 1000 万美元之后取消了。

- 增加了对系统的理解和文档化工作

对于任何一个正式的评审，它的一个好处就是会迫使评审人员为评审做好准备。在评审进行之前，评审人员需要获得构架评价的重点和用于构架表示的需求，因此就要求评审人员文档化系统的构架。许多系统并没有一个对所有开发人员都可以理解的顶层构架。构架要么太简单，要么太长。此外，在开发人员中间，经常出现他们对组件的一些假设存在误解。准备评审的过程可以揭露许多这类问题。

此外，构架评价关注于一些需要回答特殊问题的特定领域。对于这些问题的回答通常需要包含一个解释，即为什么要选择这些设计以及它们之间的关系。有一个合理的文档化的设计对生命周期后期是很重要的，这样修改所涉及的影响可以被评估。

- 对已有构架问题的检测

构架评价的第三个好处是尽早检测已有构架中的问题。在生命周期中，问题被检测得越早，修改它们的成本就越低。通过构架级检视发现的问题包括不合理的需求（或难以满足）、性能问题以及与潜在的下流修改相关的问题。例如，一个执行系统活动场景的构架评价可以大致地表明性能规格；执行系统修改场景的构架评价可以揭露可携带性和可扩展性方面的问题，如果该构架需要支持整个产品线，而不是一个单个产品的话，这个评价尤其重要。因此构架评价对产品的能力和限制提供了一个早期的透视。

- 需求的分类和优先级

构架评价的第四个重要好处是需求的验证。通过讨论和检查一个构架在满足需求方面有多好，可以使得项目所有成员对需求的理解更清晰，并且通常可以获得一个需求的优先级。当需求在独立于高层设计之外被创建时，通常会导致某些有冲突的系统属性存在于需求规格当中。高性能、安全性、容错性和低成本都想要获得，然而这通常是不可能同时获得的。构架评价揭示了这些冲突并进行了权衡，它提供了一个讨论的机会用于协商它们之间的解决方法。

- 组织的学习

把构架评价作为开发过程的一个标准部分的组织报告了被评审的构架质量获得了改进。随着开发组织学到了可能被问道的问题的类型、可能会产生的问题类型和需要用于评价的文档类型，他们自然地预先最大化了这些评审的效果。构架评价不仅仅导致评价后的一个更好的构架，同样对评价前的活动产生了帮助。长此以往，一个组织形成了一个文化，该文化提高了构架设计的质量。

总的来说，构架评价趋于改进质量，控制成本和减少预算风险。构架是所有技术判定的一个框架，这样，它对产品的成本和质量有着重大影响。一个构架评审并不保证高质量和低成本，但它指出了设计中的风险区域。其他因素，例如文档和代码的测试或质量，也会对最终系统的成本和质量做贡献。

事实证明构架还会影响已有组织的结构。例如，如果系统的部分是子承包商开发的，那么认识到两个不同子承包商开发的系统部分之间的共同部分是困难的，而构架评价可以检测这个共同性。

2. 评价技术的分类

在业界有很多技术被应用到了构架评价上，每个技术可以获得不同的信息，并且需要不同的成本。所有的这些技术基本上可以分为两个类型：通过对构架进行询问的评价技术和对构架进行大量度量的技术。我们发现询问技巧可以被用于评价任何给定质量的构架。事实上，是质量的考虑驱动了这些问题的产生（类似下面描述的以检查表的格式或场景的格式）。然而，询问技巧并不直接提供一种手段来回答这些问题。另一方面，度量技术被用于回答特定的问题。在此情况下，它们描述了特定的软件质量（例如，性能或可测量性），但该技术并不像询问技术那样使用广泛。

- 询问技术

常见的构架询问技术大约有3种：场景、调查表和查检表。这些技术在应用上彼此不同，但是它们都被用于引出对于构架的讨论并且提高构架对于其需求适应性的理解。

场景是一个特定顺序的步骤包含了系统的修改或使用。场景提供一种手段来刻画一个特定的构架是否能够适应其需求。场景可以测试我们通常所谓的可修改性、安全性、性能等。在前面我们提到的 SAAM、ASAAM 和 ATAM 都是基于场景的测试。

检查表是一个通用的或相关的开放式问题列表,可用于所有构架评价。这些问题包括对构架的产生或构架文档化的途径(例如通过询问是否存在一个已设计了的项目构架或是否使用了一个标准的描述语言),以及构架描述本身的细节(通过询问是否系统的所有用户接口特性独立于功能特性)。评价组寻找有利的反应并且追究到一定的细节,直到能够获得问题满意的答案。一个基于调查表评价过程的例子是 SEI 开发的软件风险评价过程^[271]。

“就像一个检视者使用检查表来保证结构和代码一致,软件构架师使用检查表来保证系统的所有区域之间保持平衡”^[272]。一个检查表是一个更详细的问题列表,是在有很多经验之后被开发出来的用于评价一个公共的系统集(通常是某个特定领域)。检查表比调查表更关注系统特定的质量。例如,在一个实时信息系统中性能问题会询问系统是否多次写相同的数据到硬盘,或者是否已经对处理峰值负载和平均负载给予了足够的考虑。

这三个询问技术之间有一个自然的联系。场景要用于评价特定系统的问题。评价一个相关系统族系的经验可以概括出一系列公共使用的场景,这些场景可以被转换成特定领域内的一个检查表,或者一个调查表中的一个通用项。检查表和调查表反应的是更成熟的评价实践,而场景可以反应不成熟的评价实践。一般来说,场景是作为评价过程的一个部分而被开发出来的,而检查表和调查表在项目开始前就被假设已经存在了。

● 度量技术

通过度量技术可以获得量化的结果。除了提供产生询问一个构架的问题的方法,度量技术还提供了对这些问题的答案。由于问题几乎总是先于答案,因此我们可以发现这些度量技术比询问技术要更成熟。事实上,我们所看到的度量技术的证据仅被用于回答性能或可修改性方面的问题。

度规是对构架上可观察度量的一个量化解释,例如组件的扇入和扇出。研究得最好的度量技术提供了对整个构架复杂度的答案,从中我们可以发现最有可能产生变更的地方^[273]。使用度量技术,构架评价需要关注的就不仅仅是度量的结果,还要包括使用该技术的假设条件。例如性能特性的计算假设了资源使用的模式。这些假设有效吗?

建立系统的模拟或原型可以帮助创建和澄清构架。原型的概念我们在第 14 章已经讲过。性能模型是一个模拟的例子。如果一个详细的原型或模拟仅用于评审目的,那么其创建的代价是高昂的。另一方面,这些产物经常作为正常开发过程的一部分。在这种情况下,评审中使用这些产物来回答遇到的问题就变为一个极其正常的或自然的过程。

一个已有的模拟或原型可以是一个提问技术提出的一个问题的解答。例如,如果评价组询问“你有什么证据来支持这个结论?”,一个有效的答案是一个模拟的结果。

我们可以进一步从不同的维度上来区别构架评价技术。表 15-3 给出了这些技术的一个分类总结。

表 15-3 构架评价技术分类

评价方法	普遍性	详细级别	阶段	评价对象
调查表	普遍	简单	早期	产品 过程
检查表	特定领域	各种级别	中期	产品 过程
场景	特定系统	一般	中期	产品
度规	普遍或特定领域	详细	中期	产品
原型、模拟、试验	特定领域	各种级别	早期	产品

选择哪种评价技术依赖于开发过程、组织内部构架评审成熟度和评价期间需要检查的质量。如果开发过程期间，模拟、原型或试验已经被开发，在构架评价过程中，建议使用它们来提供信息并使用模拟和原型来回答性能方面的问题。

调查表和检查表是逐步积累起来的，因此如果一个组织刚开始执行构架评价并且没有现成的调查表和检查表，那么场景是一个最好的选择。场景的开发是评价期间执行的一个活动，并且，在执行了多个评价之后，一个组织可以建立一个场景数据库并且把它们转换成调查表和检查表。甚至，如果一个组织已经有现成的调查表和检查表，那么可以使用场景来处理调查表和检查表没有覆盖的问题。也就是说，对于一个新的领域，建议首先使用场景技术，并且在随后转换成为调查表和检查表。

3. 建议的最佳实践

前面我们已经总结了用于构架评价的成本和受益，并且对5种不同的评价技术进行了分类。下面推荐一个最佳实践用于准备、执行和报告一个构架评价。

• 理解评价的上下文

(1) 评价是被计划的还是未被计划的

构架级项目评价通常出现在两种模式中的一种。在第一种中，评价被认为是项目开发生命周期中的一个正常部分。评价被预先安排了进度，内置到项目工作计划和预算中。在第二种模式中，评价并不是预先期望的，通常是因为一个项目陷入严重的问题后采取的活动，并且使用严格度量来尝试抢救先前的工作。这两种评价类型是完全不同的，它们有着不同的目标，不同的议程，服从不同的期望，并且产生不同的结果。

计划的构架评价是对项目最初方向的一个验证，是主动的行为，具有建设性。而未计划的构架评价被用于当管理者感觉到项目可能失败后采取的纠正活动，是被动的行为，具有对抗性。而作为一个最佳工业实践的建议，我们不处理未计划的构架评价。我们建议构架评价应当是开发过程的一个部分，需要预先计划和安排进度，并随后采取行动。

(2) 评价是为了发现还是用于验证

一些实践人员建议在开发过程中的某个时间点上进行一个早期的、轻量级的“构架发现评审”。这个时间点可以安排在需求已经被设置之后，并且在构架判定已经完全确定之前。在这个时间点，需求还没有在一个构架或设计方法的上下文内得到确认。这是一个理想的时间点，可以根据需求实现的可行性或成本来对需求提出异议。除了完整的构架评价

之外,可以进行类似这样的发现评审。

发现评审的主要假设是已经做出了一些设计判定,但是这些判定还没有完全绑定到一个构架上,因此可以花费很小的成本进行修改。发现评审是低成本的,但是由于绝大多数构架判定还没有被确定,因此这时候进行评审获得的好处是有限的。

(3) 评价的目的

控制评价范围是关键。一个评价可以调查许多事情,但是为了集中评价的焦点,我们建议列举一小部分清晰的目标。要关注的目标数量应当尽可能保持最小,大约3~5个。对评价来说,如果无法定义一小部分高优先级的目标,那么评审的期望可能是不现实的。这些目标定义了评价的目的,并且应当作为评价协议的一个部分明确下来。

(4) 评价系统的规模

任何评审都有一个相关的成本,并且受益应当超过成本。我们在这里建议的评价类型是适合于中等规模和大型规模的项目,而对于小项目不一定是成本有效的。我们建议的项目规模一般要超过4人年的工作量。

• 合适的人

对于一个成功的构架评价来说,有两类人的参与是非常关键的。

(1) 项目代表

项目应当包含系统构架师、每个主要组件的设计人员和系统的风险承担者。对于构架评价来说,项目构架师必须要参加,或者必须要有人能够对构架和设计发表权威判断。而对于非常大的系统,每个主要组件的设计人员需要被包含进来,以保证构架师的系统设计概念在更低的设计级别上被如实地反映和证明了。

需要确认项目的风险承担者,只有这样他们关心的问题才会被体现到评价中。风险承担者包括开发人员、经理、客户、不同类型的用户、安装人员、维护人员、协议监理和子承包商。如果系统必须和别的系统交互,那么还必须包含其他系统的代表——可能甚至包括组织的竞争对手。被选择的个人应当可以为所承担的角色说话。

(2) 评价小组

就像我们已经指出的,理想的软件构架评价组是组织内的独立的实体。评价组必须按照类似下面的方法来进行组织:

① 这个组必须被认为是公正的、客观的和受尊敬的。小组必须被看成是适合于进行评价的人组成的,这样项目成员不会把评价看成是时间的浪费,同样小组的决定就有执行的分量;

② 在评价期间,小组的成员应当做好至少100%的正常时间的投入。熬夜计划任务和在规定时间内开会经常是经常性的事情;

③ 小组应当包含对构架和构架问题很熟悉的人,并且由在构架级设计和评价项目方面非常有经验的人员来领导;

④ 小组应当至少包含一个系统领域内的专家——曾经做过被评价领域内的系统;

⑤ 小组应当包含能够处理后勤方面的人员:安排会议、定会议室、处理旅行安排、复制文档、分发事先准备好的资料、订购午餐、获得供给等;

⑥ 小组还应当包含一个图书管理员负责组织和获得与项目相关的文档;

⑦ 此外,在小组本身不具备的情况下应当可以从咨询者那里获得应用领域内的知识。

小组应当有权访问设计文档、任何工作原型、(很少情况下)源代码、评价标准知识等资源。

- 组织的期望和支持

构架评价成功的一个关键是对组织发起评审期望的共同理解,以及保证评审的资源。资深的管理者需要为评价组和项目成员设置评审的期望。这样做的一个机制是使用协议,以确定:

- (1) 谁将被告知评价完成什么?
- (2) 评价的标准会是什么?
- (3) 什么和谁会将对评价小组可用?
- (4) 对评价组和项目来说,接下来期望的是什么?
- (5) 对于项目来说,评审期望花费的最大时间是多少?

在组织内部,支持文化的一个清晰理解是关键的。组织使得构架评价有多根深蒂固?尤其是应当回答下面这些问题:

- (1) 构架评价是不是组织标准项目生命周期的一部分?
- (2) 参加评价组是否被认为是对职业的一种提高?
- (3) 组织是否认识到执行一个构架评价的标准能力是一个值得投资的核心竞争力?
- (4) 组织是否愿意投入必须的资源?
- (5) 是否存在标准的评价组织?

- 评价准备

一个成功的构架评价必须准备某些材料。包括议程、项目成员参考手册。

议程需要被关注。议程应当详细且灵活。通常,评审期间涉及到的很多信息可能不是预先能够考虑到的。如果构架评价没有采取评审会议的形式(例如,评价可以是一个独立组准备的报告),那么评价的目标和评价的标准必须被清晰地表达。例如,“我们的系统应当被集成”这个目标应当被表述得更特定,如“我们的系统应当在不超过10天的工作量之内完成”。

对于评审指导的参考手册应当提供给项目成员。理想情况下,可能是一个评审人员会问到的问题检查表。如果调查表或者检查表将要在评价期间被使用,它们应当在评审前提供给开发组。

4. 建议总结

构架评价的主要建议总结如下:

- 和外部评审人员一起进行一个正式的评审

外部评审人员具有一定的客观性和独立性,不带有作者骄傲的色彩。他们提供一个机制用于引入构架专业技术。

- 选择最有利的评价时间

建议进行评价的时间是在需求已经被建立之后,并且有一个建议的构架要评审。一个构架评价不能在建议的构架存在之前进行。就像前面所说的,AT&T有一个评审类型叫“构架发现评审”,它的确是一个在构架评审前检查需求合理性的活动,但是这个评审类型回答的是不同的问题。

在任何开发工作中,随着时间的推移,构架的修改变得越来越受限制,因此越来越昂贵。尽可能早地执行构架评价可以减少构架修改带来的成本耗费。

- 选择一个适当的评价技术

在前面我们列举了5个不同的评价技术,被分类为询问技术和度量技术。场景首先出现作为一种通用建议的技术,可以作为组织开始采纳构架评价的实践。其他询问技术派生于场景,并代表了组织的构架评价成熟度。

原型是一个高成本的技术,如果仅为了评审而开发原型是不恰当的。原型在需求的分析过程中是一个受欢迎的技术,但是那些原型通常并不能代表任何最终的系统,因此它们的构架不应当作为构架评价的基础。对于性能评价,建议使用模拟技术。但与原型一样,成本是该技术的一个限制。

度规对帮助理解设计的可修改性是有帮助的,但是度规对要执行的修改类型做了假设。一般来说,在度规之下的假设不是很清晰,因此我们不强调把它们作为一个评价技术。

- 建立一个评价协议

由于我们建议评价由一个外部评审组来执行,因此双方理解评审将完成什么,不将完成什么是重要的。一个正式的协议仅仅是一种机制来保证对评审的范围有一个相互的理解。

- 限制被评价质量属性的数量

通过关注一个预先同意的有限的质量数量,评价组可以保证开发组织有它的优先顺序,并且所有参与方都理解那些优先级。当问题(超出范围)产生时,由于要研究的质量是清晰的,并且已经被公布,因此很容易把评审拖回轨道。如果缺乏对范围的限制,那么评审可能会迷失方向,并且不会执行它被设计的任务。

- 坚持系统构架

对于一个成功的开发来说,有一条规则是:要有一个人,他能够对整个系统都有所了解(不一定每个都了解得很详细)。Fred Brooks 写到:

“我比以前更相信:概念的完整性是产品质量的核心。有一个系统的构架是通向概念完整性的最重要步骤……在教了软件工程实验20多次以后,我开始坚持认为,就像4个人这样的学生小组也应当选择一个经理和一个独立的构架师。”^[274]

15.3 SDL 及相关测试

15.3.1 SDL 介绍

SDL(Specification and Description Language)是ITU-T推荐使用的一种“实时系统的规格描述语言”,详细资料可以参考Z.100标准。应用的主要领域是实时系统行为方面的规格,以及这类系统的设计。电信领域在这方面的应用包括:

- 交换系统中的呼叫及连接处理(如呼叫处理、电话信令、计费);
- 一般电信系统中的维护和故障处理(如告警、自动故障清除、例行测试);

- 系统控制(如过载控制、更改及扩充过程);
- 操作和维护功能、网络管理;
- 数据通信控制;
- 电信业务。

当然,任何实物,只要其行为的功能规格可以用离散模型规定,也就是说,此实物可用离散消息与其环境进行通信,就可以用SDL来规定其行为的功能规格。

SDL是一种丰富的语言,它不但可以用于高层非形式(和/或形式化的不完全)的规格描述,而且还可用于半形式化和具体的规格描述。在使用该语言时,用户应根据所要描述的通信层次和环境选用SDL的不同部分。依赖于功能规格的应用环境,在功能规格的来源与目标之间,可能还会有许多方面需要进一步的共同理解。

使用SDL,可以半图形、半文本地定义特定类型的嵌入式系统的功能描述。这种方法的高度正式性,使得SDL工具有可能生成和测试完整的嵌入式应用。国外大量软件评论家认为,SDL的这种正式方法,应该推荐扩展应用到嵌入式系统之外。事实上,如SDL的消息顺序图,已经被普遍地应用到面向对象技术CASE工具中,如Rational Rose,在交互式图形部分广泛采用了这种技术。

15.3.2 SDL 基本概念

一个SDL设计中包含了系统、环境、功能块、信道、信号、信号路由、进程、过程、定时器、服务等多个元素。

1. 系统

一个SDL系统就是用SDL规格所描述的一个具体物理存在,也就是我们目前从SDL语言的角度所关注的一切,对于目前不用SDL语言描述的,都作为SDL的环境看待。也就是说,一个具体的物理系统,如果只用SDL描述了其中的一部分,那么这部分就是一个SDL系统,而该物理系统的其他部分,对该SDL系统来讲,都是它的环境。

SDL系统通过信道与环境连接。从理论上讲,SDL系统只需要一条双向信道与环境连接,但实际上,为了描述方便,常常为环境的每个逻辑接口提供一条信道。

每个系统可以划分为多个功能块,功能块之间通过信道相连,系统与环境之间的信道实是连接到系统内的功能块的。各功能块相对于其他功能块而言是独立的,功能块之间以及功能块与环境之间的通信是靠发送信号来实现的。

2. 环境

在介绍SDL系统的概念时已经提到,SDL的环境是目前不用SDL规格描述的外界。由于外界这个范围是极其广大的,所以有必要对作为SDL系统的环境做进一步的界定。通常情况下,外界除了与SDL系统有交互作用的部分以外,还有许多与SDL系统没有任何作用的部分。对于这部分不与SDL系统发生任何关系的外界,在SDL的语义上都不能成为SDL系统的环境。其实SDL系统和SDL环境都是我们在构造一个完整的系统时所关注的,对于那些我们根本不关心的外界,自然可以忽略。

在与 SDL 系统有相互作用的部分, 由于存在直接作用和间接作用, 又可对 SDL 环境做进一步界定。对与 SDL 系统没有直接作用的部分也可排除在 SDL 系统的环境范畴之外。简单来讲, SDL 系统的环境就是与 SDL 系统有相互通信的, 而又没有用 SDL 规格描述出来的那部分实体。

3. 功能块

SDL 的功能块是在 SDL 系统内的部分功能集合。通常情况下, 功能块是一些关联较为密切的功能的集合。功能块的划分可以有多种依据: 把部件定义得其大小规模便于处理; 能与实际的软硬件划分相适应; 与自然的功能划分相一致; 把交互作用减小到最小等。由于出发点和功能划分的依据不同, 一个 SDL 系统可以有多种的功能块结构。

一个功能块又可根据需要划分为子功能块, 以便在不同程度上对系统进行描述。子功能块与进程处在同一级, 他们都通过一定的路由与信道相连。

功能块所包含的功能最终是由在功能块内的进程来实现的。一个功能块又可由多个不同的进程来分别实现不同功能。

4. 信道

信道是系统的功能块之间功能块和环境之间进行通信的手段。一条信道可以单方向地(单向信道)或双方向地(双向信道)将一个功能块连接到另一个功能或者连接到环境。通常信道是一种功能性实体, 可用来表示特定的信息通路。事实上通过划分信道, 可以形式地规定每个信道的行为。

一个信道规格为每个指定的方向给出一个信号表, 列出能由该信道在该方向上传递的全部信号标识符。此信号表作为一种手段, 用以保证由信道一端某一进程发送的每个信号, 都能被位于信道另一端的功能块中的进程所接收。

5. 信号

信号是 SDL 系统内以及 SDL 系统与环境进行信息交互的载体。信号定义一般包括信号名和类别表, 类别表表示该信号传递的值的类型的表。在具体的应用中, 类别表由实参组成的参数表替代。

由于 SDL 是基于 EFSM 的, 所以, 无论信号名还是信号参数表都蕴涵着一定的信息

6. 信号路由

信号路由与信道相似, 都用来表示通信路径。它们可以使用在功能块层次上, 也可使用在进程层次上。同信道一样, 信号路由可以是单向的, 也可是双向的, 但不能进行划分。

在功能块层次上, 它们用来表示一种在功能块诸进程之间或在进程和功能块环境之间通信的手段。这里的功能块环境就是通向或离开该功能块的一条信道。

在进程层次上, 信号路由用来将进程分解为服务子结构的场合。此时信号路由把服务相互连接起来, 或把服务连接到进程的信号路由上。

当一个信号被传到通向功能块边界的一条信号路由时, 该信号即被交付给连接到该信

号路由的信道。当一个信号从一条信道到达功能块,且该信道连接到一条或多条信号路由时,该信号即传向能够传递该信号的信号路由。

7. 进程

进程是一种扩展的有限状态机,它规定一个系统具体的动态行为。也就是说,只有到达了进程级,系统详细的动态行为才得以描述。

进程一般处于等待信号状态,当收到一个信号时,进程做出响应,执行特定的动作,在执行完相应的全部动作以后,进程进入下一个状态等待信号或者停止(杀死)。

一个进程可以在系统创建时就存在,或者作为另一个进程的子进程被创建。进程的生存周期可以是无限制的,也可在一定的条件下停止(收到一个停止信号)。进程在被创建以后,只能在一次状态跃迁过程中执行一条停止动作自我杀死,若其他进程想停止某个进程就需要向该进程发送一条特定的停止信号,收到该信号后的动作在进程的定义中给出。

一个进程定义规定了一类进程。可以创建一类进程的多个实例,这些实例可以同时存在,各自独立并发地执行。同时存在的进程实例最大值在进程定义时就给出。

8. 过程

在SDL中之所以引入过程的概念,是出于以下考虑:

- 使进程的结构构成可进入一些细节层次;
- 允许用一个单项来表示一个可以孤立地加以对待的多个项的复杂组合,以保持规格紧凑;
- 允许定义常用的项的组合,便于重复使用。

过程定义可以包含在进程定义、服务定义和过程定义中,并只能在它的定义范围内可见。

9. 定时器

SDL中的“定时器”模型是一些元进程,它们在遇到请求时可以向进程发送信号。

要使用定时器必须先对进程中对定时器进行声明。“SET”和“RESET”操作是用来激活定时器的。“SET”操作即设置定时操作请求在到达某一特定时间时发送出到时信号,而RESET操作即清零操作则是取消设定的定时。

在SDL中用函数NOW表示当前的绝对时间,通过NOW加上表示相对时间的表达式,可以将相对时间转换为绝对时间。

10. 服务

服务是由进程提供的一种“功能”,它是表示该进程“子行为”的进程定义的一部分,这样的“子行为”是进程总行为的一部分。

设置服务概念的目的在于将进程定义划分但又不引入并行的可能性。也就是说,将进程划分为服务而不是重新划分为多个进程,既满足了将进程的功能做进一步划分的目的,又减少了因进程并发运行而带来的种种麻烦。之所以说进程并发会带来种种麻烦,是因为进程并发运行时,各个进程都可以接收信号,发送信号,从而可能会存在同时的状态跃迁,

这种不同行为并发造成的组合将会增加控制的复杂度。而将进程划分为服务,由于每个进程每次只能从外部接收一条信号,所以每次只能有一个服务收到信号,从而减少了控制的复杂度。另外,进程与进程之间为了共享(读和写)数据,又不得不进行复杂的信号交互,而利用服务则可以在进程内通过公共数据完成,从而减少信号交互带来的复杂度。

一个服务就是一个拥有自己独立空间的有限状态自动机,我们可以将它理解为一个“小进程”。服务体的具体实现与进程体是相同的,具体的描述将在后面给出。

15.3.3 SDL 结构

SDL 包含 4 个部分,可以使用表 15-4 来表示。

表 15-4 SDL 的 4 个成分

属性	SDL 构件
框架(Architecture)	系统、功能块(System, block)
行为(Behavior)	进程(Process)/服务(Service)
通信(Communication)	信号及信道(Signal and Channel)
数据(Data)	抽象数据类型(Abstract Data Types)

一个完整的 SDL 系统必须至少包含一个功能块。一个功能块必须至少包含一个进程或一个功能块。最后,该树的树叶是进程。功能块和进程在一个功能块中不能同时出现。系统的所有行为描述在进程中。它们的关系可以用图 15-8 来表示。

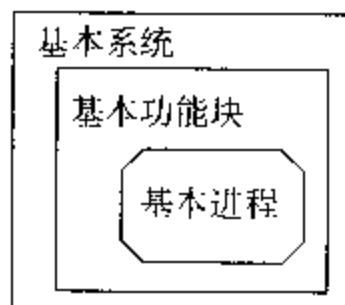


图 15-8 SDL 框架示意图

SDL 使用的结构化设计思想,它的构架描述了不同层次的抽象,包括:

- 系统与外部世界的关系;
- 功能块交互;
- 进程交互;
- 进程和过程中的行为描述。

它们之间的通信主要通过以下来定义:

- 信道和信号路由;
- 信号和信号参数(数据)。

图 15-9 至图 15-13 给出了一个 SDL 从系统到进程/服务的实现样例。

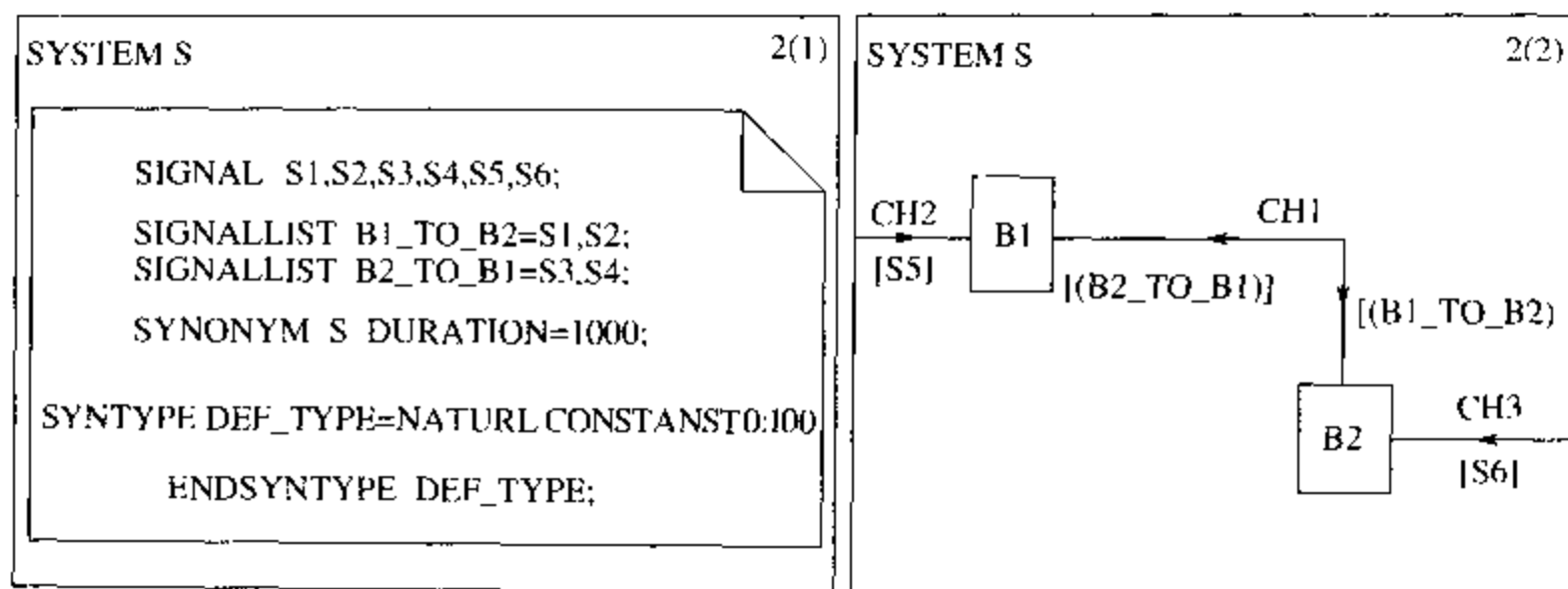


图 15-9 SDL 系统图

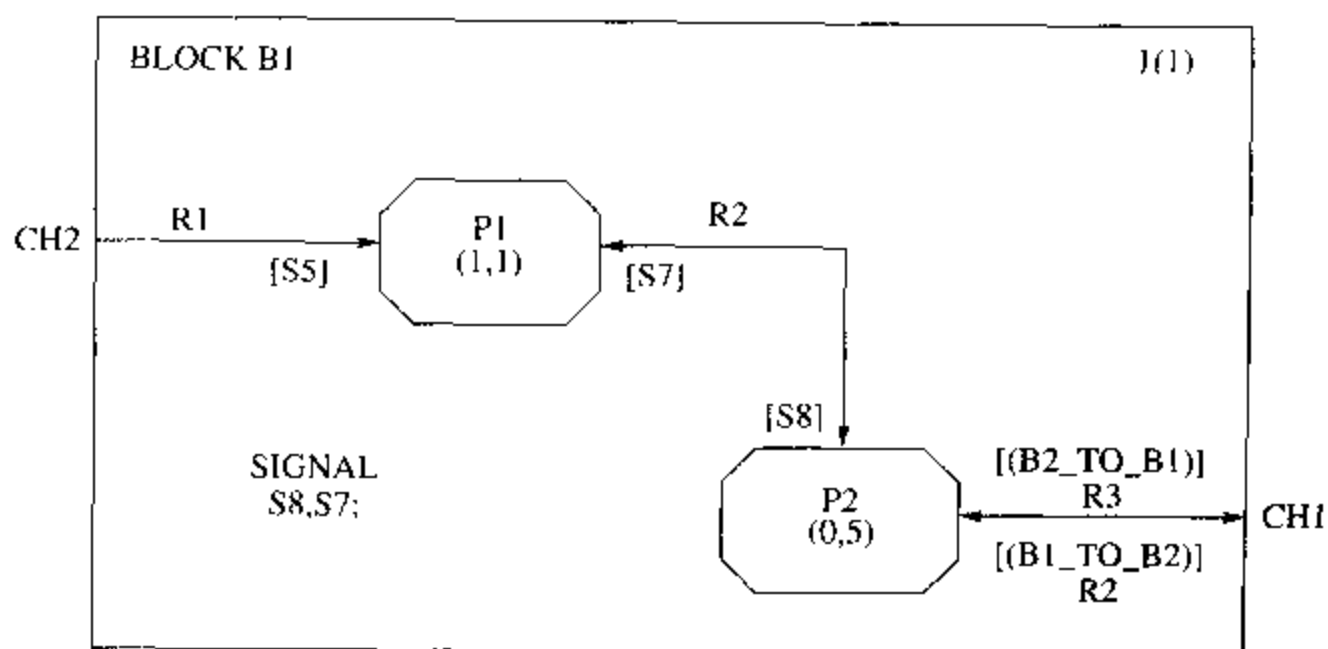


图 15-10 SDL 功能块图

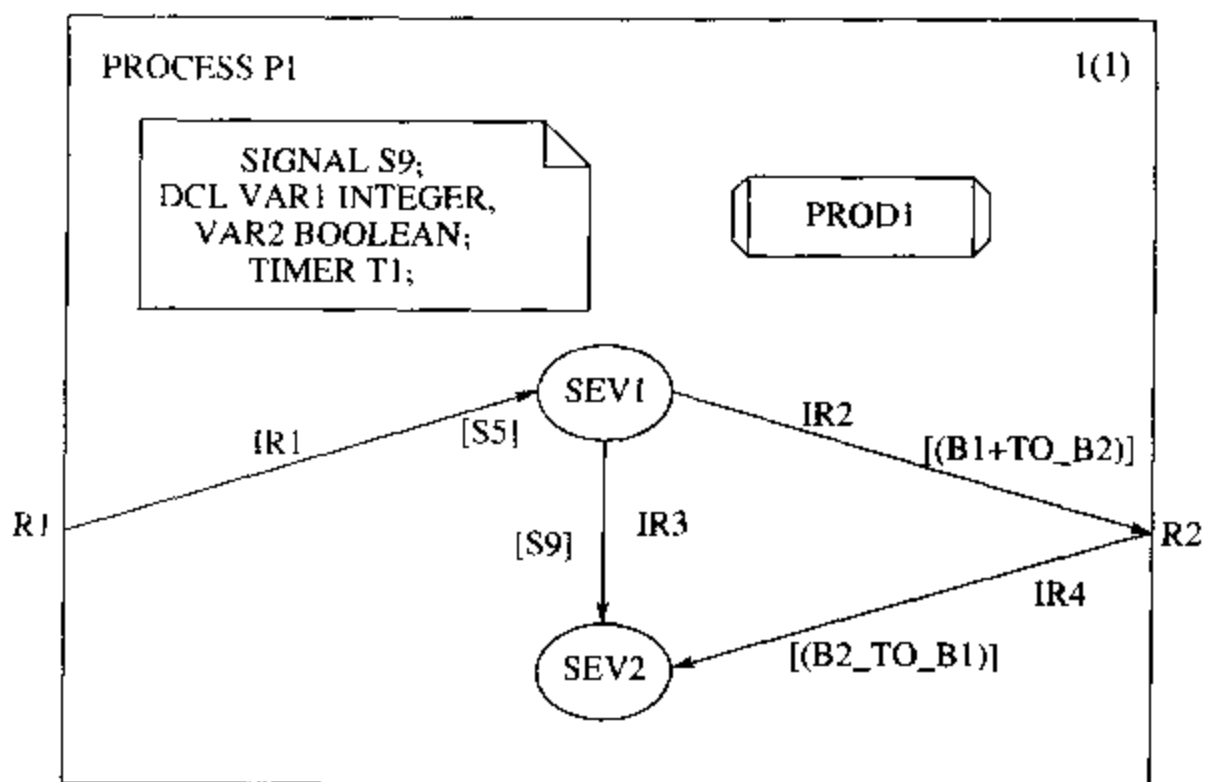


图 15-11 SDL 进程图(含服务)

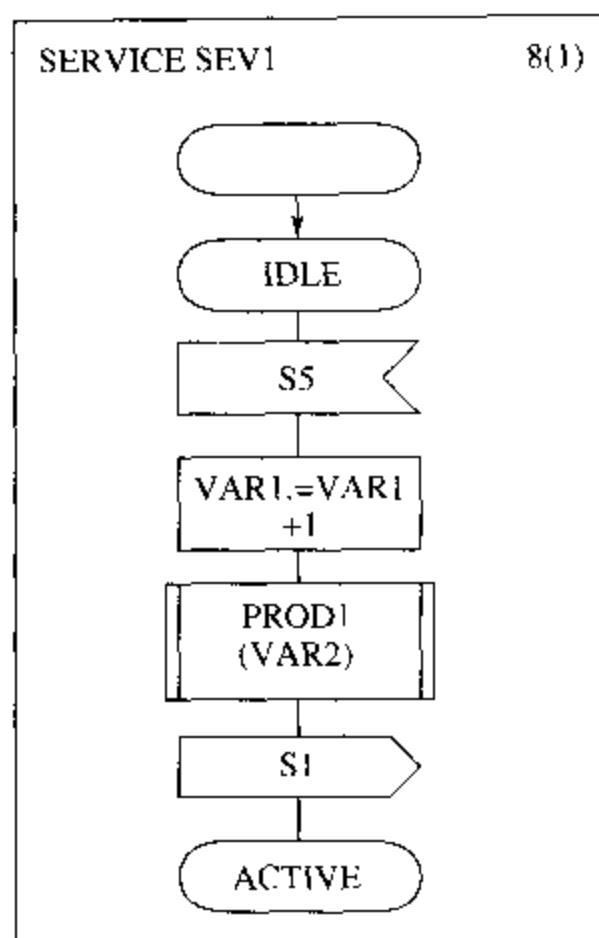


图 15-12 SDL 服务图

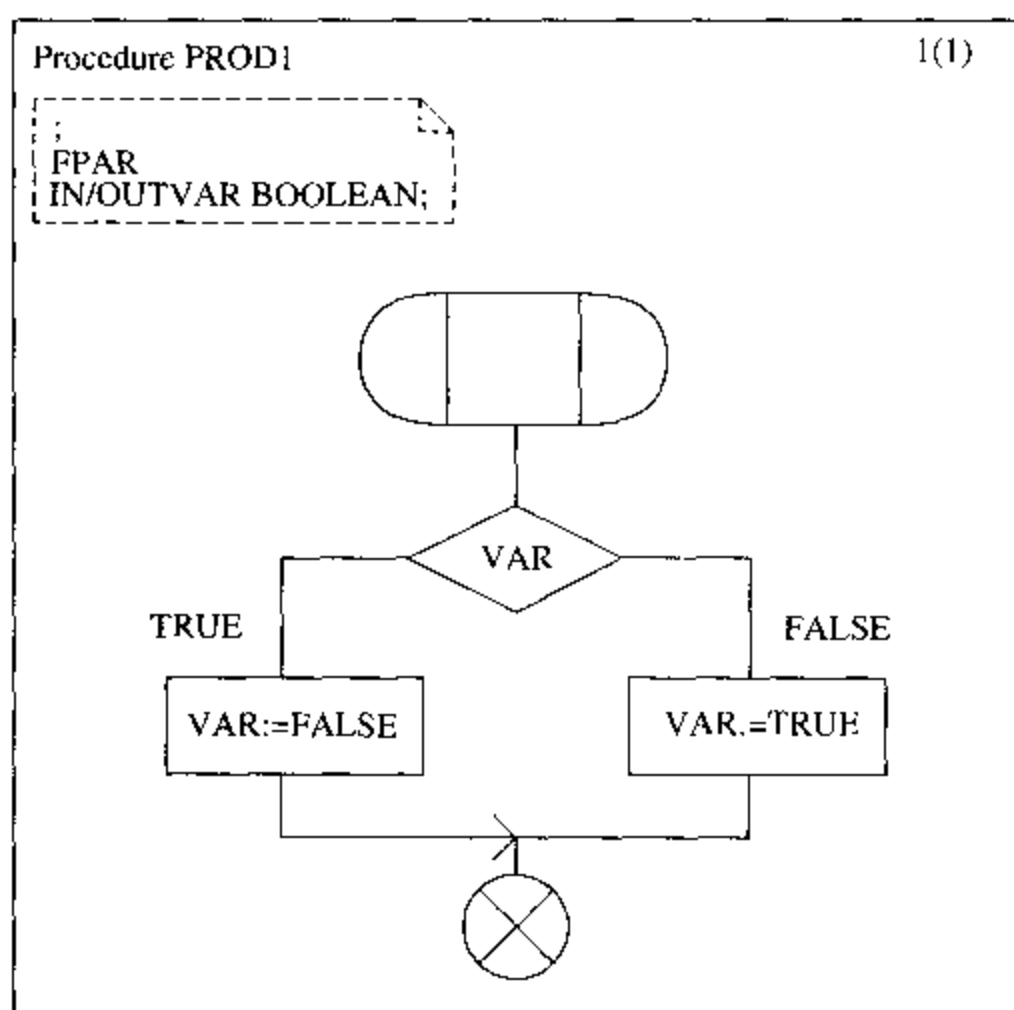


图 15-13 SDL 过程图

15.3.4 SDL 测试

SDL 是一种规范的设计描述语言，针对这类语言的验证可以借助一些工具。Telelogic 公司的 SDT 工具中提供了两个验证 SDL 的小工具：Simulator 和 Validator。

1. SDL Simulator

SDL Simulator 是一个 SDL 的调试模拟工具，它可以提供对整个 SDL 系统过程的模拟，也可以对 SDL 系统中的一部分进行模拟(某个功能块、进程)。

Simulator 提供了两种操作方式——菜单式和命令行式。菜单(或工具条)在用户接口窗口的左边，命令行则在窗口右侧。在命令行窗口内，提供了同 Doskey 相同的功能。其界面如图 15-14 所示。

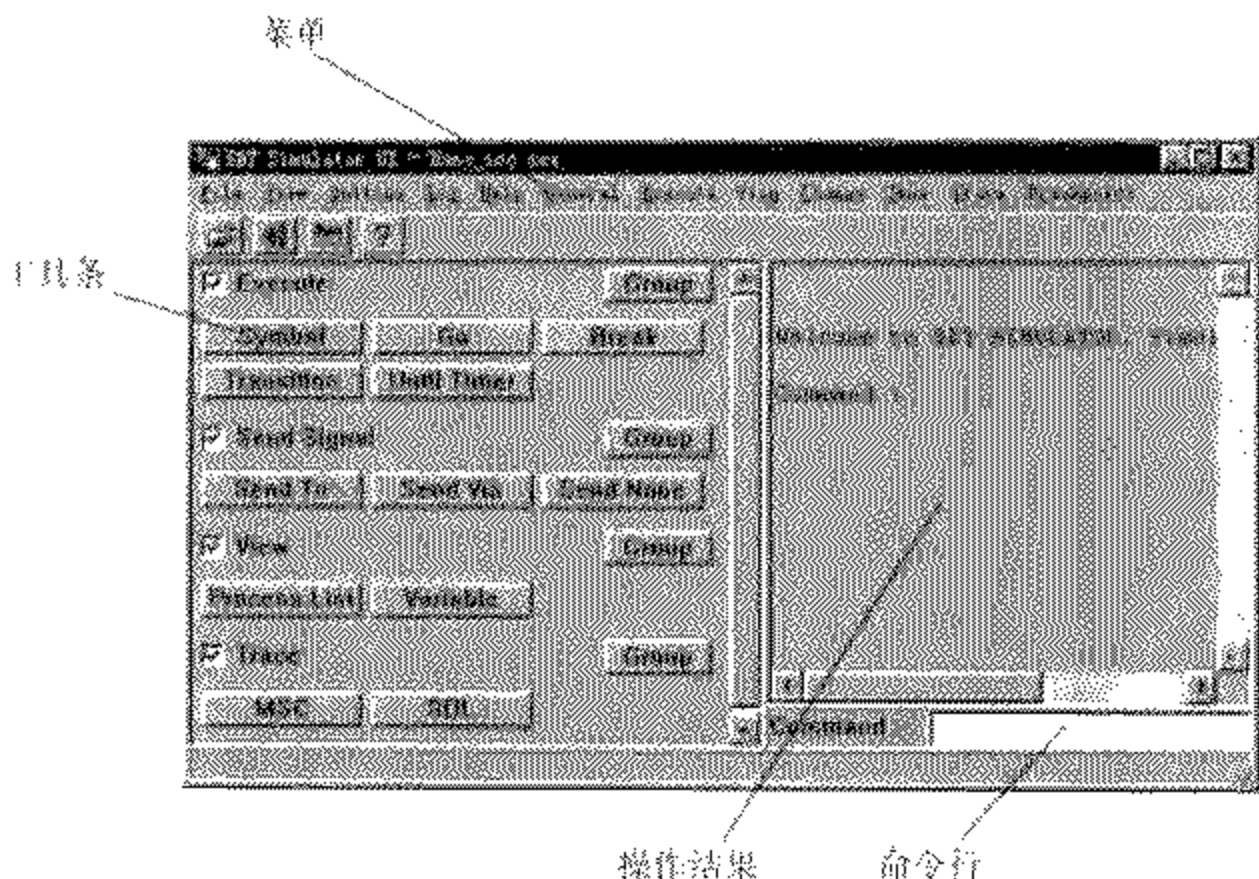


图 15-14 Simulator 界面

Simulator 的命令行方式为了方便用户的使用，提供了智能的识别方式，即：你可以输入命令行的缩写，如 Output-to 可以缩写为 o-t、ou-t、o-to 等。如果缩写造成了歧义，在操作结果窗口内会提示你可能匹配的所有命令的全称。另外，当只输入命令，而没有输入参数时，会出现参数窗口供你选择。所有的命令可以在 General 菜单的 Command 项中查到。

工具条共有 4 个组，单击各组的 Group 按钮，可以添加、编辑、删除本组内的工具条。

- Execute 组

Symbol: Step-Symbol, 单步执行，每次执行 SDL 的一个符号。

Go: Go, 全速执行，直到阻塞，即必须有一定的外界条件发生才能进行下去，如等待环境输入新的信号。

Break: 中断目前的操作。

Transition: Next-Transition, 执行一次状态跃迁。

Until Timer: Proceed-To-Timer, 全速执行，直到有超时信号产生。

- Send Signal 组

Send To: Output-to, 发送一个信号到指定的进程实例。

Send Via: Output-Via, 通过指定的信号路由发送一个信号。

Send None: Output-None。

- View 组

Process List: List-Process, 列出目前存在的进程实例(名称、Pid、状态)。

Variable: Examine-Variable, 查看指定进程内的变量值。

- Trace 组

MSC: Start-Interactive-MSC-Log 1, 显示模拟过程的 MSC 描述。

SDL: Set-GR-Trace, 显示模拟过程在 SDL 系统 GR 描述中的历程。

在模拟时, 如果系统很大, 一旦在某些地方输入了错误的命令或者想改变运行环境, 还可以通过菜单 Change 来改变一些值, 如改变状态、变量, 删除一个已送出的信号, 设置/取消定时等。

在某些情况下, 我们进行模拟的很多过程是重复的, 为了避免每次都重复输入命令, Simulator 提供了一种批处理方法, 我们可以自己编写批处理文件, 也可以将我们的操作过程记录下来, 成为下次使用的批处理。具体做法是, 在操作命令前, 打开菜单 Log, 选择 Start Command Log 项, 输入批处理文件名, 开始保存操作过程, 操作完成时, 选择 Log 菜单中的 Stop Command Log 项。也可用文本编辑方式输入自己想操作的命令, 并将文件保存为 .com 文件。每次想运行批处理文件时, 打开 Execute 菜单中的 Script, 选定批处理文件即可。

模拟过程中, 还可以使用 Coverage Viewer 来查看我们模拟所经历状态、分支在整个 SDL 系统中所占比例, 以及各经历过的状态、分支的统计数字。使用方法: 选择菜单 Show 中的 Coverage 项。Coverage Viewer 是验证 SDL 系统时常用的一个工具。

2. SDL Validator

SDL Validator 是以某一种算法为基础, 对整个的 SDL 状态空间进行遍历, 来检查 SDL 系统中有没有死锁, 不可达等错误。验证的机理还是比较复杂的, 在此就不介绍了, 有兴趣的读者可以参考相关的文献。这里我们只介绍 Validator UI 的简单使用方法, 帮助大家有一个感性认识。

Validator UI 窗口与 Simulator UI 窗口的风格一样, 不同之处只是各自的工具条和菜单不同。具体看图 15-15。

- Explore 组

Bit-State: 使用 Bit State Space 算法, 对 SDL 系统的状态空间进行遍历。在进行这样的工作时, 有两个重要的参数需要我们进行定义。一个是遍历时最大的搜索深度, 一个是使用的 Hash 表的大小。搜索深度表示在进行状态遍历时一个行为树所能达到的最大深度。

在一次 Bit-State 完成之后或被中断, 会弹出一个报告窗口(默认设置, 也可在命令行中使用 Define-Report-Viewer-AutoPopup 命令或在主菜单 Option1 中设置), 报告产生的错误。在报告窗口中双击各错误项, 会自动弹出对应本错误的 MSC 过程描述。另外, 在操作结果窗口中也会给出一些统计信息, 如图 15-16 所示。

这些统计信息中, 常用到的有下面几项。

(1) No of reports: 错误报告个数。

(2) Truncated paths: 因到达设定的搜索深度而被截断的行为树的个数。

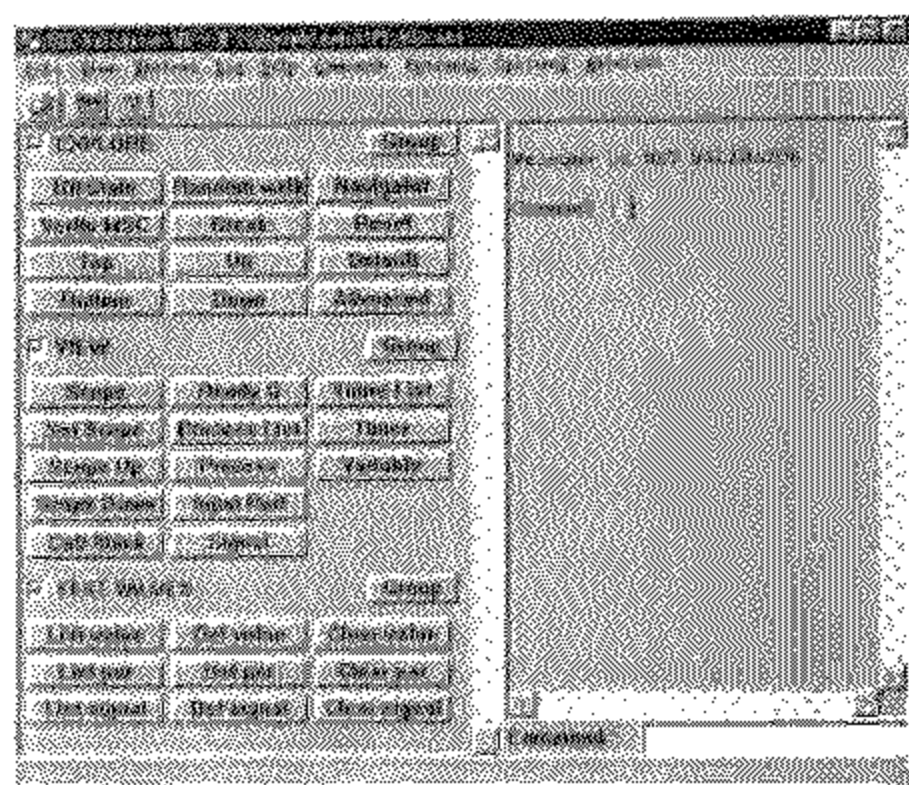


图 15-15 SDL Validator 界面

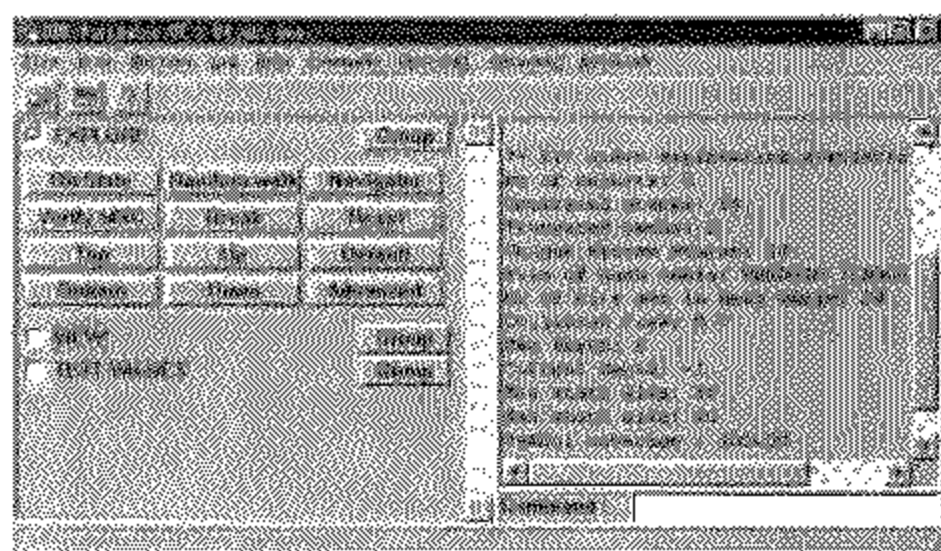


图 15-16 SDL Validator 统计信息

(3) Collision risk: 冲突率。因 Hash 表的存在而发生冲突的比例。

(4) Symbol coverage: 符号覆盖率。遍历所覆盖的 SDL 符号率。

Random Walk: Random Walk 采用的是这样的一种状态空间自动遍历方法, 它从当前的系统状态(System state)出发进行遍历, 当在某一状态下, 同时存在多个跃迁时, 随机选取一个跃迁进行下去。Random Walk 也有两个比较重要的参数设置, 一是 Search depth, 它的含义同 Bit State 的 depth 是相同的, 一是 Repetitions, 它表示从系统状态开始, 可以进行随机状态跃迁的次数。Random Walk 的操作结果与 Bit State 相似。

Navigator: 激活 Navigator 工具。Navigator 是 SDL 提供的一种交互式工具, 用来在 SDL 系统的行为树中“航行”。值得注意的是, 在 Validator 和 Navigator 中描述的行为树, 与系统设定的各种数据类型的值有关。这点我们会在后面看到。

Navigator 的使用方法很简单, 在 Navigator 窗口内, 有两层方框组成, 上层用来进入上个状态, 下层用来进入下一个状态。使用方法是, 我们可以选择一个方框(方框内有关于跃迁的信息, 如输入信号, 信号发送者等), 如图 15-17 所示; 然后双击该方框, 窗口内的内容就根据本次跃迁改变。这种操作过程同模拟很相似, 只不过每次的输入都是预先设

定好的。

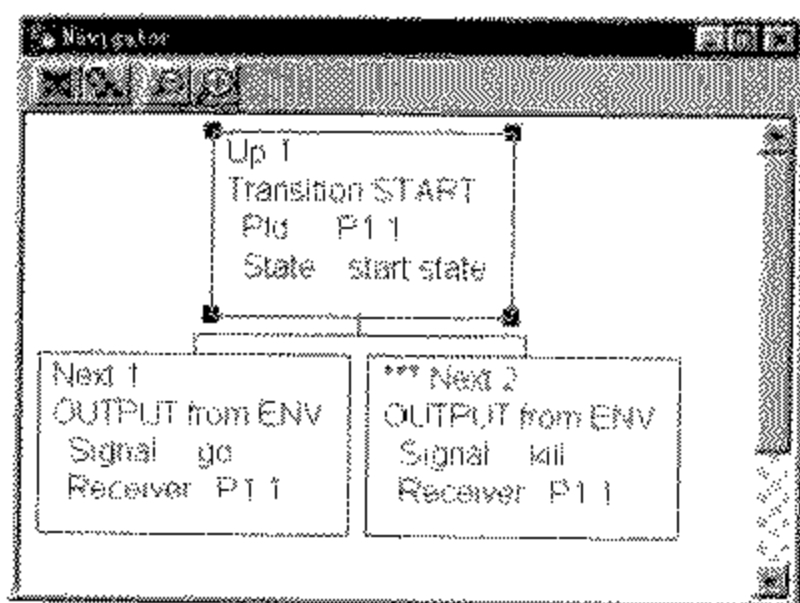


图 15-17 Navigator

Verify MSC: 根据提供的 MSC 图中的行为轨迹对 SDL 系统进行遍历。

Break: 中断正在进行的遍历。

Reset: 复位 Validator 到初始态。清除报告, 各种用户定义的规则, 复位所有的选项。其实此时 Validator 会读取 Valinit.com 文件。

Top: 行为树回到顶端, 即系统状态。

Bottom: 移动到行为树的末端。

Up: 将当前状态向上移动 n 级, n 为参数。Up 的默认值为 1。状态的改变我们可以在 Navigator 中很清楚地看到。如果 n 过大, 状态将停留在根状态下。根状态可以通过 define-root 改变。

Down: 将当前状态向下移动 n 级, n 为参数。Down 的默认值为 1。若当前状态下有多个跃迁可以选择, Down 无法进行下去。

Default: 使用默认选项。

Advanced: 提供了 Define-Scheduling All; Define-Priorities 1 1 1 1 1; Define-Max-Input-Port-Length 2; Define-Report-Log MaxQueueLength Off 等几条命令的集合。

Define-Scheduling All 表示每个状态下, 允许进程实例就绪队列中的所有实例运行。与之相对的是 Define-Scheduling First, 只允许就绪队列中的第一个进程实例运行。默认值为 First。

Define-Priorities 1 1 1 1 1 用来定义几类事件的优先级(共 5 级, 1、2、3、4、5)。默认值为内部事件 1、环境输入 2、定时输出 2、信道输出 1、自发跃迁 2。

Define-Max-Input-Port-Length 2: 定义最大输入端口队列长度为 2。默认值为 3。

Define-Report-Log MaxQueueLength Off: 关闭有关 MaxQueueLength 的报告输出。默认值为打开。

• View 组

Scope: 查看当前域。

Set Scope: 设定域。即设定我们观察的范围(哪个进程实例, 服务实例)。只有在设定域之后, 查看变量、信号和输入队列才有意义。

Scope Up: 域上移一级。从服务上移到它隶属的进程。

Scope Down: 域下移一级。从进程下移到进程实例内的服务, 如果有多个服务存在, 必须指定服务。

Call Stack: 显示定义的域范围内的进程/实例的名称、类型、状态。

Ready Q: 列出就绪队列中的进程实例。

Process List: 列出指定进程激活的进程实例。如果没有指定进程, 则所有激活的进程实例被列出。

Process: 列出指定域内进程实例信息。包括 parent、offspring、sender 的值和进程实例本身的 PID, 状态。

Input Port: 列出指定域范围内进程实例的输入队列内的信号信息。

Signal: 列出输入队列指定信号(根据信号在输入队列内的登录号)的参数。

Timer: 列出激活定时器队列中指定定时器(根据定时器的登录号)的参数。

Timer List: 列出当前域内所有目前激活的定时器。

Variable: 查看变量或形参值。如果没有给出变量名, 所有的变量和形参值被列出。

• Test Value 组

这组按钮对应的命令主要是修改、显示有关各数据类型、信号参数和信号的信息。这是我们在进行 Validate 时常用的一组按钮, 通过设定、修改各种数据类型值, 增加信号, 可以获得很好的验证效果。

List Value: 列出预定义的数据类型的值。所有从环境输入的信号的参数值就决定于这些预定义的各数据类型值。在没有定义新的数据类型值时, 只列出需要使用的数据类型的值。这些值在由系统默认给出时, 一般是这种数据类型两端的值。

Define Value: 定义数据类型的新值。通过定义数据类型的新值, 可以获得我们需要的输入信号。

Clear Value: 清除所有定义的数据类型值。

List Parameter: 列出用户定义的信号参数。

Define Parameter: 定义信号参数。

Clear Parameter: 清除所有定义的信号参数。

List Signal: 列出所有用户定义的信号。

Define Signal: 定义信号(含信号参数)。

Clear Signal: 清除用户定义的信号。

验证的基本机制就是在给出外界环境输入的前提下, 排列组合各种输入对 SDL 系统的行为树进行遍历, 以期望获得在随机模拟各种输入情况下 SDL 系统的处理情况。所以, 定义好数据类型值, 信号参数值和信号是很重要的。

值得注意的是, 如果环境信号输入参数值设置不当, 有些行为分支可能永远也走不到。可以通过修改定时信号的优先级来模拟各种的超时处理。另外, 在定义搜索深度时, 不宜太大, 也不宜太小。太小时, 验证很快就结束了, 没有深入, 很多分支都还未走到, 意义不大。搜索深度太大时, 在一个行为树分支上耗费的时间太长, 也不能获得全面的遍历效果。我们可以设定一个适中的搜索深度进行遍历, 在结束后, 再通过修改系统根状态, 在某些不易达到的行为分树上进行遍历。修改系统根状态可以先通过 Navigator 到该

状态，再使用 Define-Root Current 命令来修改。

3. TTCN

TTCN(The Tree and Tabular Combined Notation) 是一种用于协议一致性测试的规格表述，能够以对协议数据单元和抽象业务原语观察和控制的观点来进行抽象的表示。它是 ITU-T X.292 标准推荐的测试方法。TTCN 这种记法的设计目的为：

- 提供一种将抽象测试用例用标准化测试套来表示的记法；
- 提供一种与测试方法、层、协议无关的记法；
- 提供一种可以反映 X.290 系列建议所定义的抽象测试方法(ATM)的记法。

在抽象测试方法中，一个完整测试套被看作分级的：测试组、测试例、测试步和测试事件。TTCN 为反映测试实例在分层结构的位置而提供了一整套的命名规则。TTCN 认为最基本的测试事件为：接收和发送，抽象业务原语(Abstract Service Primitives)，协议数据单元(Protocol Data Units)和定时器事件。

TTCN 支持两种数据类型定义：一种是 TTCN 表格形式，一般用于较低层次协议的数据类型定义；另一种是 ASN.1 类型，表达层以上层协议数据类型定义通常要用到。

TTCN 提供了两种格式的记法，一种是人可以直接读懂的表格形式(TTCN.GR)，用于描述对应用 ISO 协议的产品一致性测试套表述。另一种是机器可执行形式(TTCN.MP)，用于计算机内部或者计算机系统间处理。这两种记法是同等的，由于有标准的支持，TTCN 编辑环境有树形和表格相结合的固定格式。TTCN 测试套源文件的存放格式是文本的，后缀为 .mp。

对于使用 SDL 设计的系统，使用 TTCN 进行测试具有很大的优势。Telelogic 提供的 ITEX 直接可以在模拟层测试 SDL 系统，它们之间的测试关系可以通过图 15-18 来表示。

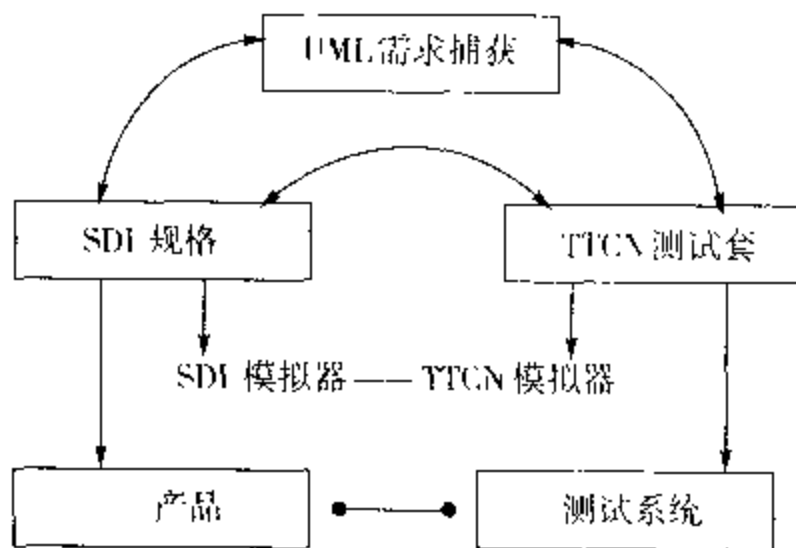


图 15-18 TTCN-SDL 测试关系

由于需要涉及到很多的背景知识和 SDL 及 TTCN 语法知识，因此在此就不详细讲解了，感兴趣的读者可以参考以下标准。

X.290: General Concepts 通用概念

X.291: Abstract Test Suite Specification 抽象测试套规格

X.292: The Tree and Tabular Combined Notation TTCN(1992,1998 修订)

依据上面制定的测试体系结构采用 TTCN 来编写测试套。

测试套：测试例的集合，通常可分为多个测试组，用来完成一种或多种协议的动态一致性测试(包括能力测试和行为测试)。

TTCN：为 OSI 一致性测试套的编写定义的一种标准化测试套记法。

X.293：Test Realization 测试实现

生成 PTS，并完成 Meaning of Test

X.294：Requirements on Test Laboratories & Clients

一致性评估过程中的需求，进行 IUT 的测试，并出具报告

X.295：Protocol Profile Test Specification

X.296：Implementation Conformance Statements(ICSs)

ICSs 一致性实现陈述，来源于协议规格，并用于 IUT 的测试

Telelogic 公司 TAU 的帮助文档里有关于 SDT 和 ITEX 方面的详细使用手册，用户可以进行参考。

15.4 本章小结

设计是一个承上启下的过程，它把抽象的用户需求转换成具体的可实现的系统结构，这是一个需要创意的过程，有人把它理解成一种艺术。正因为如此，这个过程也是最易于产生风险的过程。如何把握好设计的质量成为软件工程领域内的一项课题，本章在这方面做了一些探索，总结了业界在该领域的一些经验，提出从静态的评审到动态的测试等多种手段。

目前业界在构架设计评审方面使用最多的是基于场景的评审方法，最基本的方法是 SAAM。在该方法的基础上可扩展出很多新的方法，例如本章中介绍的 ASAAM 以及 ATAM。

SDL 是一种基于结构化设计的设计描述语言，主要应用在嵌入式领域。目前关于 SDL 验证方面有许多可以应用的工具，包括 Telelogic 的 Simulator、Validator 以及 ITEX。TTCN 作为一种 ITU-T 推荐的协议一致性测试方法，可以和 SDL 进行无缝连接。因此，对于 SDL 设计的系统，使用 TTCN 作为测试描述语言是非常好的。

第16章 同行评审

在第14章和第15章谈到关于需求测试和设计测试方面的一些方法和经验,在这些测试中,使用最多的手段就是同行评审。同行评审属于静态测试中的手工测试范畴(参考第2章)。实践证明,同行评审有着比动态测试更高的缺陷发现效率,本章将重点介绍同行评审的种类、评审的方法,以及在开发过程中如何安排各种评审的经验,通过本章的学习,你将了解以下内容:

1. 了解同行评审包含的4种类型以及它们之间的关系;
2. 描述同行评审的一般过程;
3. 明确正规检视小组各成员角色的职责;
4. 描述正规检视的过程。

16.1 基本概念

同行评审(Peer Review)是一种通过作者的同行来确认缺陷和需要变更区域的检查方法。需要进行同行评审的特定产品在定义项目软件过程的时候被确定并且作为软件开发计划的一部分被安排了进度。

在该定义中,同行(Peer)是一个项目组成员,他被分配执行指定产品的一个同行评审。根据特定的同行评审过程,他为他承担的角色负责。该同行应有一定的开发经验以及足够的产品知识,以便能够理解评审中的产品。而作者(Author)对需要进行同行评审的产品负责并且需要提交材料给同行评审组。

在这里我们一直提到了产品,这个概念与习惯上的理解不同,你可以把它理解成最终产品的组成部分。它是在软件开发或维护过程中产生或需要的一个可交付的文档。产品的例子包括:需求文档、设计文档、软件代码和单元测试产品、用户/操作手册、支持手册、计划文档和过程文档等。

同行评审是一个总的概念,在实际执行时有不同的组织形式,有严格的,也有松散的。根据具体情况的不同,可以把同行评审分为走读(Walkthrough)、技术评审(Technical Review)和正规检视(Formal Inspection)^[220]。在有些资料中,也把管理评审(Management Review)包含到同行评审中^[142]。但笔者认为管理评审不应当被包含到同行评审中去,因为同行评审的关键在于技术专家和开发同行的参与,在这类评审中,管理者的参与会改变评审过程并且歪曲参与者的客观性^[222]。而管理评审的主要参与者是管理者,目的是确保项目的进展和资源的合理分配。表16-1给出了这几类评审的区别^[223]。

表 16-1 同行评审类型

分类	管理评审	技术评审	正规检视	走读
目标	确报进展 建议纠正措施 确报资源合理分配	评估与规格和计划 的一致性 确报变更完整性	检测和识别缺陷 验证解决方案	检测缺陷 检查替代方案 学习讨论
控制决策的制定	管理团队制定活动 计划 决策在会上制定或 是建议的结果	评审小组请求技术 专家按照建议执行	小组选择预先定义 的产品部分 缺陷必须被排除	由作者作出决策 变更是作者的特权
变更验证	由其他项目控制活 动验证	技术专家验证评审 报告	主持人验证修改	由其他项目控制活 动验证
小组规模	2 人或更多	3 人或更多	3 ~ 6 人	2 ~ 7 人
参加者	管理者 技术专家 同行	技术专家和同行	同行	技术专家和同行
领导者	通常是负责经理	通常是主任工程师	经过培训的主持人	通常是作者
评审对象规模	较高 取决于会议对象的 状态	较高 取决于会议对象的 状态	比较低	比较低
发起人	项目负责人	软件单元负责人	作者以外的人	一般是作者
数据收集	根据标准、政策的 要求进行	非项目正式要求	正式要求	非项目正式要求
输出报告	管理评审报告	技术评审报告	缺陷列表 缺陷总结 检视报告	走读报告
数据库人口	任何进度变更都必 须输入项目跟踪数 据库	没有正式要求	记录缺陷统计、特 点、严重程度、会 议属性	没有正式要求

16.2 同行评审的一般过程

一个同行评审的过程可以使用图 16-1 来表示。

16.2.1 计划阶段

1. 分配角色和职责

不管哪种类型的同行评审，都会应用到下面这些角色和职责。在走读、技术评审和正规检视中使用到的特定角色将在相应的章节中进行描述。

角色 项目管理者

职责 项目管理者对下面这些活动负责。

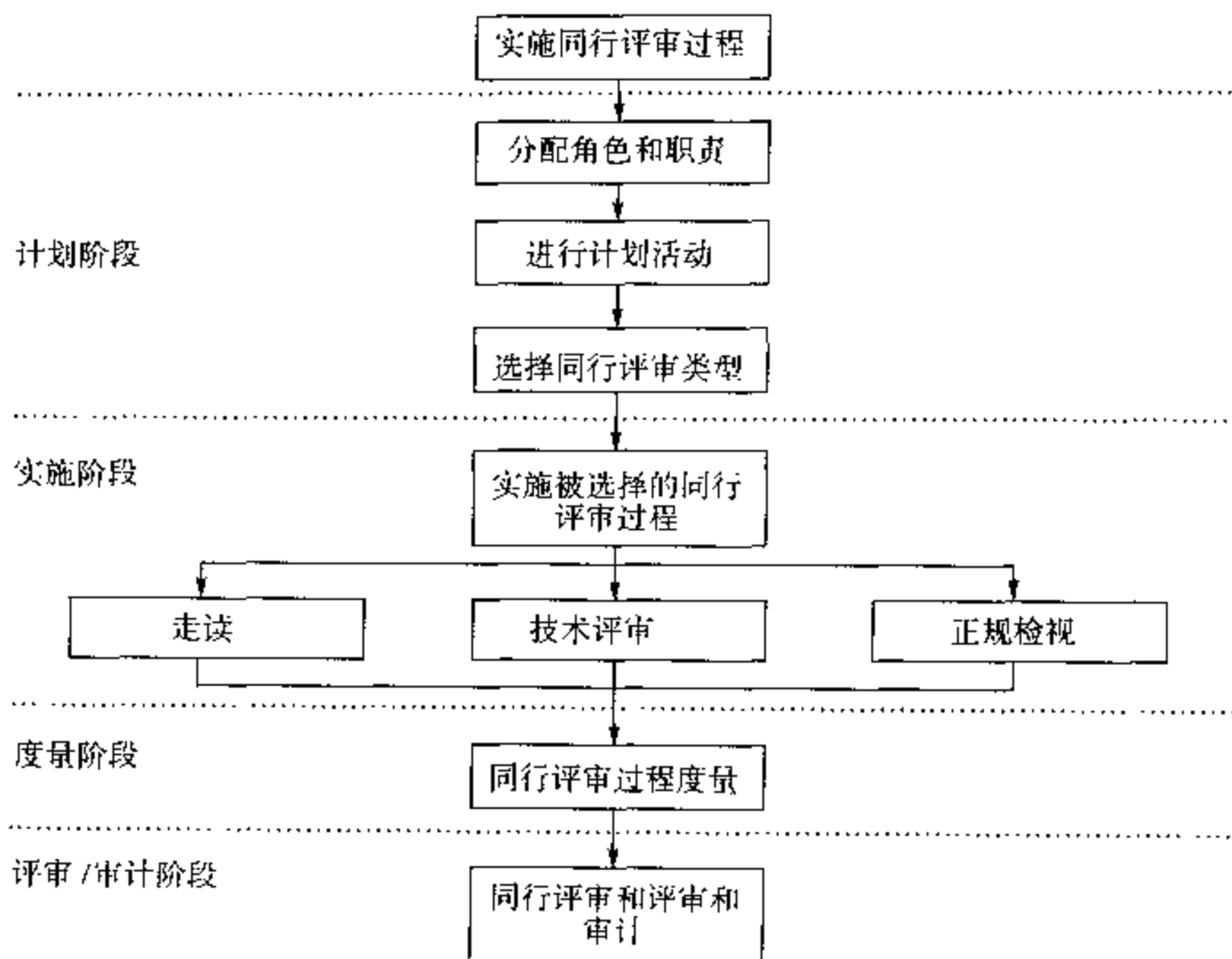


图 16-1 同行评审过程

- (1) 批准同行评审策略；
- (2) 保证项目软件开发计划包含每个产品的同行评审以及同行评审的特定类型；
- (3) 提供必要的时间、人员、预算和设施来计划、执行和管理同行评审；
- (4) 保证评审被执行，根据项目里程碑事件对同行评审结果进行评审，保证进行产品的再工作；
- (5) 在使用同行评审过程中对项目组成员进行必要的培训。

角色 项目成员

职责 项目成员对实施同行评审政策负责并且参加同行评审方面的培训。在同行评审过程中，作为一个参与者有一定的开发经验以及足够的产品知识是非常关键的。

角色 作者

职责 对需要评审的产品负责，并且及时提交材料给同行评审组。

角色 同行

职责 同行(Peer)是一个项目组成员，他被分配执行指定产品的一个同行评审。根据特定的同行评审过程，他为他承担的角色负责。该同行应有一定的开发经验和足够的产品知识，以便能够理解评审中的产品。

角色 软件质量保证人员(SQA)

职责 SQA 独立的评审同行评审过程的实施，保证被选择的同行评审类型适合于被评审的产品。最终的结果被报告到管理者那边。

角色 软件工程过程组织(SEPO)

职责 由于需要在组织内满足用户的需求, SEPO 维护和更新这个过程文档。这个过程文档将根据组织用户输入的能够反映过程改进的数据以及组织的经验和教训进行更新。SEPO 在同行评审过程上给项目提供帮助, 并且可以提供适当的培训。此外, SEPO 维护正规检视度量数据库。SEPO 接受到正规检视日志和数据并登录到正规检视数据库中。这些数据被用来和正规检视过程有效性进行比较。

2. 进行计划活动

对任何同行评审类型来说, 需要描述下面这些计划活动。这些活动可以被认为是同行评审过程的入口条件:

- 产品标准被定义。像类似于可读性、可维护性、一致性、功能性、可测试性等产品标准被充分地定义。软件开发和文档标准已经被确认。软件开发和文档标准包括 MIL-STD-498, 数据项描述和商业标准(如 IEEE)。在产品的软件开发计划中应当引用到产品标准、开发标准和文档标准。
- 检查表被定义。用于进一步定义一个产品标准的检查表已经被开发和评审过。同行评审使用的检查表可以帮助检查和分类潜在的产品缺陷。
- 评审产品可管理。产品被分割成可管理、可评审的单元, 这样同行评审会议可以在 1~2 个小时之内被执行完毕。
- 软件开发计划定义被评审的产品和使用的评审过程。
- 同行评审被计划。根据已定义的软件开发计划, 同行评审被计划并安排了进度。
- 人员被培训。参与到同行评审中的人员获得需要的培训, 包括同行评审的目标、原则和方法。此外, 被委派的同行评审负责人接受了如何领导同行评审的培训。
- 资源和资金被确认并计划。资源和资金被确认、计划并文档化在软件开发计划中。
- 同行评审过程被文档化。
- 用于同行评审的度量机制适当且可用。同行评审执行和结果的数据被收集、维护和报告。度量被建立以确定同行评审活动的状态。
- 用于同行评审的产品已经就绪。对于同行评审来说, 一个产品被认为是就绪的情况可以是: (1) 作者声明产品就绪并且准备提交产品; (2) 配置管理申明产品就绪; (3) 项目就绪标准被满足。

3. 选择同行评审类型

表 16-2 给出了一个如何选择同行评审类型的指导^[220]。在该表格中, 第一列显示了要进行同行评审的产品类型, 包括了需求文档、设计文档等项目过程产物。第一行给出了根据产品复杂性建议的情况, 共分为 3 级: 简单、中等、复杂。

表 16-2 同行评审选择举例

产品	技术复杂度		
	简单	中等	复杂
软件需求	正规检视	正规检视	正规检视
设计	技术评审	正规检视	正规检视
软件代码和单元测试	走读	技术评审	正规检视
验证测试	技术评审	技术评审	正规检视
用户/操作手册	走读	技术评审	正规检视
软件文档, 例如: 版本描述文档, 软件产品规格书, 软件版本描述等	走读	走读	走读
计划文档	不应用	不应用	不应用
过程文档	不应用	不应用	不应用

16.2.2 实施被选择的同行评审过程

根据上一节选择的同行评审, 实施走读过程、技术评审过程或正规检视过程。详细的过程内容将在 16.3 节, 16.4 节和 16.5 节进行介绍。

16.2.3 同行评审过程度量

执行和使用度量来确定同行评审活动的状态。度量的例子包括: (1) 执行的同行评审的数量与软件开发计划进行比较; (2) 花费在同行评审上的工作量与计划的工作量进行比较; (3) 被评审的产品数量与项目的开发计划进行比较。

收集的数据包括: (1) 被评审产品的标识; (2) 产品的规模; (3) 评审组的规模和组成; (4) 评审会议的长度; (5) 发现和修正的缺陷类型和数量; (6) 返工的工作量。

16.2.4 同行评审的评审/审计

为了保证同行评审的正确执行和效率, 需要对同行评审的执行进行评审和审计。其目的是要验证已计划的评审被执行了。评审或审计应当被计划并安排到软件开发计划中。如果软件质量保证功能可用, 这一般会是 SQA 的职责。此外, 进行验证以保证项目成员为他们承担的角色进行了培训。同行评审的审计/评审可以揭露同行评审参与者是否为同行评审做了准备, 同行评审是否按照规定的过程被执行等。一次审计还可以揭露同行评审数据的提交、收集和报告是否完整、精确, 且及时。

16.3 走读

走读的目的是要评价一个产品，通常是软件代码。走读一直以来都与代码检查联系在一起，其实走读也可以应用到别的产品（如：结构设计、详细设计、测试计划等文档）上。走读最主要的目标是要发现缺陷、遗漏和矛盾的地方；改进产品；考虑可替换的实现方法。走读还有其他一些目的，包括：技术的交换、参与人员的技术培训、设计思想的介绍等。走读可以指出代码中效率和可读性方面的问题，设计或不可测设计规格中的组件应用问题。走读是同行评审三种形式中最自由的一种形式，其成员包括作者和一个或多个评审人员。

16.3.1 过程目标

发现缺陷、遗漏和矛盾的地方；改进产品；考虑可替换的实现方法。

16.3.2 特定的角色和职责

走读组(Walkthrough Team)：一次走读有一个或多个成员。每个成员需要在走读之前评审任何输入的资料，并且在走读期间参与评审以保证目标被满足。

作者可以要求或指派一个小组成员执行记录员的职责。如果一个记录员被指派，记录员就需要负责记录走读期间作出的所有说明，包括发现的问题、样式方面错误、遗漏、矛盾、改进建议或者替换解决方法。

16.3.3 输入

走读的最小输入包括：

- 准备用于走读的产品；
- 可应用的标准或检查表以帮助产品的评审。

16.3.4 人口标准

产品已经就绪，可以进行走读。

16.3.5 过程

步骤1 计划走读会议

作者完成下面工作：

- 通过选择一名或多名人员组成走读组。如果需要有一个记录员，作者指派这个角色；

- 安排走读会议时间和地点；
- 分发所有必须的材料给评审人员，例如用于产品评审的标准、检查表及被评审产品。作者确定是否需要进行一次产品介绍以便参与评审的人员对产品有一个大致了解。作者应当允许评审人员有足够的时间来评审材料。

步骤2 评审产品

评审人员负责为走读做准备并且如果必须的话，需要完全熟悉标准、检查表和其他任何提供的用于走读的信息。评审人员评审产品并且必须准备在走读会议上讨论他们对产品作出的评注、建议、问题和红色标记。

步骤3 进行走读

作者走读整个产品。小组成员可以对产品提出问题，并且/或者记录他们关心的事情。如果指派了一个记录员，记录员记录评注和决定，这些内容将包含到走读报告中。

在走读结束时，评审人员可以建议进行下一次走读。

步骤4 解决缺陷

作者和评审人员解决走读中发现的问题。无法解决的问题需要提交到项目领导那边寻求解决。

步骤5 记录走读

作者至少需要记录评审人员的名字、被评审的产品、走读的日期、缺陷、遗漏、矛盾和改进建议列表。有多种方法来记录被执行的走读，例如软件工程记事本或者软件开发文件夹。

步骤6 返工产品

根据走读中的记录，作者返工产品。

16.3.6 出口标准

- 在整个产品被详细走读过后，走读就完成了；
- 所有缺陷、遗漏、效率问题和改进建议已经被记录了；
- 作者把执行的走读记录到了适当的记事本、文件或文件夹中。

16.3.7 输出

走读过程的输出是一个被执行走读的记录，包括所有缺陷、遗漏、效率问题和改进建议的记录，以及根据走读被纠正了的产品。

16.4 技术评审

技术评审是由一个正式的组对产品进行评价。它确认任何与规格和标准不一致的地方或者在检查后给出可替换的建议，或包含这两者。技术评审的严格程度没有像正规检视那么严格。技术评审的参与者包括作者，以及产品技术领域内的专家。

16.4.1 过程目标

技术评审的过程目标是要评价一个产品并给管理者提供以下证据：

- 产品遵从项目的计划、标准、指导书和需求规格；
- 针对产品的变更被恰当地实施，并且仅影响那些被变更确认的系统区域。

16.4.2 特定的角色和职责

评审组长：评审组长负责组织技术评审。这包括与评审相关的管理任务，并且保证评审按照有序的方式进行。评审组长还要对提出的一些技术评审细节问题负责。如果有一个活动项数据库，评审组长还需要跟踪活动项直到关闭。

记录员：记录员负责记录评审组做出的发现（例如：缺陷、不一致、遗漏和模糊）、决定和建议。如果有一个活动项数据库，记录员还要负责把这些活动项登录到数据库中。

技术评审组成员：每个组成员负责准备评审并且保证评审满足它的目标。此外，组成员还需要把所有建议使用一种适合于管理者采取适当行动的格式进行记录。

管理者：管理者负责及时的根据评审组建议采取行动。

16.4.3 输入

技术评审最小输入包括：

- 准备评审的产品；
- 项目的计划、标准、指导书和需求。

16.4.4 入口标准

产品已经准备好可以进行技术评审。

16.4.5 过程

步骤1 计划技术评审会议

评审组长完成下面工作：

- 确定评审组，如果需要可包含适当的管理人员；
- 安排和宣布技术评审会议时间和地点；
- 分发所有必须的材料给评审人员，评审组长应当允许评审人员有足够的时间来评审材料并且有时间来参加评审会议；
- 确定是否需要一个产品介绍会议。如果需要一个介绍会议，介绍会议可以和评审会议放在一起，也可以单独进行。

步骤2 评审产品

评审人员检查产品和相关资料。

步骤3 进行技术评审

在评审会议期间，整个组评审软件产品，评价产品与可用的指导书、规格和标准的相关状态，或者评价可替换的问题解决策略。尤其是，评审组要执行下面这些任务：

- 检查评审中的软件产品并且验证它符合必须遵从的规格和标准。所有与规格和标准偏离的地方都被记录了下来；
- 记录技术问题、相关建议和解决问题的责任人；
- 确定其他必须表述的问题。

步骤4 解决缺陷

评审组长和作者解决评审中发现的问题。当缺陷很多或很严重时，评审组长应当建议对缺陷修改后的返工产品进行一次额外的技术评审。

步骤5 产品技术评审会议细节

这些细节包括：

- 评审参与人员的姓名；
- 被评审的产品；
- 技术评审的日期；
- 未解决的产品缺陷列表；
- 管理问题列表；
- 活动项当前责任人和状态；
- 评审组提出的任何关于如何安排为解决问题和缺陷的建议。

步骤6 返工产品

作者返工产品。

16.4.6 出口标准

- 技术评审细节中被确认的所有问题已经被描述；
- 技术评审的所有细节已经被提出和分发。

16.4.7 输出

会议细节和返工后的产品。

16.5 正规检视

软件正规检视是在软件开发过程中进行的，发现、排除软件在开发周期各阶段存在的错误、不足的过程，是一种软件静态测试方法，其生存周期为软件的开发周期，应用于开发过程中产生的（非阶段性）软件文档和程序代码。

正规检视不同于其他类型的同行评审，它遵循一个严格的过程，人员经过培训，检视

过程有评估标准,正规检视针对实际的产品或者半成品,目的是发现存在的错误。正规检视的参加者来自开发部门、测试部门、质量保证部门或者用户。正规检视比其他评审更严格、更容易组织、效率也更高。正规检视不能代替阶段评审、静态检查或者测试。

正规检视具有以下基本特点:

- 按规定程序和时间计划进行;
- 在进入开发下一阶段前,尽可能多地发现产品中存在的错误;
- 由具有正规检视知识和了解被检视对象的审查人员,完成对检视对象的审查过程;
- 以3~7人组成的小组完成具体工作,组织者对整个过程的负责;
- 检视对象的开发人员(作者)必须参加;
- 检视人员的任务、职责明确;
- 在软件开发各阶段过程中的某些点进行;
- 以获得项目管理、质量评估的数据和检视过程本身的改进为目的,而不是评估人的能力。

16.5.1 正规检视小组

正规检视小组的规模很小,是由设计、开发、测试、质量等不同部门中工作性质相关的职员中特别关心产品的那些人组成。同时使用者或用户也可作为小组成员。最小正规检视小组的人数是3个人,一般正规检视小组的人数4~7个不等。规模大的正规检视小组主要是正规检视高级开发阶段的文档,规模小的正规检视小组主要检视具体技术的实现。根据需要,检视小组可以增加检视者。最佳组合的检视小组应当具有不同技术领域的经验,这对检视过程非常重要。具有这样知识和经验组合的检视小组,每个检视者都从他们自己的观点出发检查产品,有益于发现很隐蔽的错误。“协作”是正规检视的特色,一个检视者的想法经常会引起另一个检视者的其他想法,这是正规检视过程的规范性表现。

1. 成员角色和职责

(1) 组织者

组织者主持、引导正规检视的运行过程,全面负责正规检视的效果。因为这个角色对正规检视过程非常重要,对正规检视组织者的培训也要比其他检视者更重要,培训的范围也更广。除了修改错误阶段,组织者一直参与正规检视过程所有阶段的工作。因此在正规检视过程中组织者的投入相当多的时间,并且要进行专门训练,组织者通常是由开发部门挑选和培训,并且担负指定开发项目的正规检视工作。组织者的责任包括组建检视小组、分配检视小组的角色、领导正规检视过程。组织者的主要功能是保证在检视过程中控制检视小组的情绪,并且在检视会议期间不要讨论开发者的缺点。组织者负责收集正规检视报告所要求的正规检视数据。

组织者的职责是:

- 与开发者确定检视对象;
- 组建检视小组,并通知小组每个成员;

- 向小组成员发放相关资料；
- 为每个成员分配检视任务；
- 确定开始检视的时间；
- 确定是否举行检视对象的介绍会议；
- 主持正规检视的两次会议：检视对象介绍会议和检视会议；
- 确认对发现的问题，开发人员已正确修改；
- 确认检视对象中无法确认问题已妥善处理；
- 确定本次检视是否可正常结束；
- 收集检视资料(如方法、规范等)，总结检视经验；
- 与检视人员要协力合作，解释、满足检视人员提出的合理要求；
- 确认检视结果已归档；
- 通过主持正规检视，不断改进检视过程；
- 为阶段检视(阶段评审)提供数据。

组织者必须具备的条件：

- 明白正规检视的目的；
- 明白正规检视在开发过程中的作用和重要性；
- 能对检视对象进行客观的检视，不进行人身攻击；
- 具有作为开发者或检视人员参加过正规检视的经验；
- 领导能力必须得到检视小组成员的认可。

组织者在正规检视开始前的检查项目：

- 检视对象是否已准备好；
- 所需的检视资料是否已经拿到；
- 所有资料是否已准时分发给每个检视者；
- 检视者是否已收到发放的相关资料；
- 所有检视者是否已同意本次检视的时间安排；
- 检视会议的会议室是否已安排妥当；
- 必要的硬件设备(如投影仪等)是否已准备妥当。

组织者在检视会议上的检查项目：

- 检查每个参加人员是否进行充分准备；
- 正规检视是否已取得了一致的目标；
- 正规检视是否按计划进行；
- 每个检视者对自己的工作是否尽职尽责；
- 检视者的兴趣是否开始衰减；
- 每个检视者是否都已陈述自己的观点；
- 是否已形成了最后的检视结果；
- 是否出现了情绪性的争吵，影响了检视工作的进行；
- 检视者对检视结果的意见是否一致；
- 所有检视者对检视结果是否已真正明白。

组织者在检视会议结束之后的检查项目：

- 本次正规检视是否成功；
- 是否达到了预期的效果；
- 若不成功，找出原因所在；
- 是否形成了正确、及时的检视报告；
- 所有参加人员对检视报告是否满意；
- 检视对象是否得到了公平、妥当的处理；
- 是否已成立了处理检视对象中无法确认问题的小组；
- 相关部门和人员是否得到了本次检视的有关结果和信息；
- 开发者与检视者是否从本次正规检视中得到了益处；
- 通过本次正规检视，是否提出了改进下一次正规检视的有效建议。

(2) 开发者

开发者是检视对象的生产者。主要负责提供关于检视对象的资料并且回答检视者的问题，保证区分理解性和实际的错误。开发者也是正规检视的检视者。开发者要负责修改所有在正规检视会议期间已确认的严重错误，以及在时间和条件许可下改正一般错误。

开发者的职责：

- 准备检视对象，以便于制订检视对象的查检表、检视标准；
- 审阅开始本次正规检视的准则；
- 与组织者协商管理本次检视，制订检视计划；
- 提供产品介绍会议上所需的资料；
- 在检视小组会议上，回答检视人员提出的问题；
- 认真改正检视人员发现并已确认的问题；
- 使组织者能够确认，所有问题在后续阶段都已正确修改；
- 与检视者要协力合作，解释、满足检视人员提出的合理要求。

(3) 检视者

检视小组的每个成员都被认为是一个检视者，可以兼任不同的角色。检视者在正规检视的准备阶段和在检视会议阶段都具有发现产品错误的责任。检视者作为检视小组的成员，还要承担其他的职责，担任组织者、开发者、讲解员和记录员等适当的角色。

正规检视的候选人主要是在过去、现在和将来的产品生命周期中直接参加产品开发的人。例如，在设计方案检视中，合适的检视者可以是完成产品需求的分析员、将要编写代码的程序员和系统接口部分的设计者。这个角色有一个例外情况是编写代码的程序员，不适合作为正规检视那部分代码测试过程的检视者，因为编写代码的程序员可能希望改变测试过程，使得他们编写的代码能够顺利通过正规检视。参与产品开发并对产品感兴趣的任何人都可以作为潜在的检视小组成员，包括系统工程师、测试工程师、软件质量保证人员、系统管理员和系统使用者，还有用户等。检视者的来源不局限于软件开发部门的内部，在外部门或者外单位中具有专门经验的，能够促进正规检视的效果的人员都可以作为检视者。

检视者的职责：

- 根据检视计划，安排、分配自己的检视时间；
- 全面阅读相关资料；

- 参加检视对象介绍会议和检视小组会议；
- 完成分配的检视任务；
- 完成检视任务、参加第 3 小时会议、帮助开发人员修改错误、支持组织者和开发者的工作；
- 保持良好的职业素质，以工作为出发点，不对某人的能力进行评估；
- 全力去发现检视对象中的问题；
- 提高产品开发的专业水平；
- 为作好下一次正规检视积累经验。

(4) 讲解员

讲解员的责任是在正规检视会议期间讲解检视对象，引导检视小组对产品进行彻底审查。在产品生命周期中下一阶段的用户、使用者是讲解员的最佳候选人。阅读和解释检视对象的过程，会使用户在产品移交之前非常熟悉产品。讲解员也要承担一般检视者的任务。

讲解员的职责：

- 为详细解释检视对象做好准备；
- 在检视会议上，从头至尾将检视对象解释给检视小组；
- 在对问题进行讨论时，需提供清楚、易懂的资料；
- 对理解困难的部分要作好标注；
- 将检视过程中发现的与上一开发阶段有关的问题，返回到上一开发阶段；
- 与检视者要协力合作，解释、满足检视者提出的合理要求。

(5) 记录员

记录员负责在检视会议期间，在正规检视审查列表上记录所发现的每个错误。正规检视审查列表包括错误的出处、错误的简单描述、错误的分类和发现错误的检视者。记录员必须记录已确认或无法确认或者是上游文档的错误。记录员也要承担一般检视者的任务。

记录员的职责：

- 理解检视表中列出的问题分类标准；
- 在检视会议上，记录问题登记表中的每一个问题；
- 检视小组会议结束后，与组织者共同完成最终的检视错误列表；
- 与检视者要协力合作，解释、满足检视人员提出的合理要求。

2. 角色兼职原则

检视小组的每个成员都有自己规定的任务和职责，并要求尽职尽责。在具体操作时，可根据人员配置，遵循以下兼职原则：

- 讲解员和记录员；
- 开发者和记录员；
- 组织者和记录员。

以下人员兼职是不允许的：

- 讲解员和开发者；
- 开发者和组织者；

- 组织者和讲解员。

16.5.2 正规检视过程

检视过程包括7个阶段，7个阶段之间有严格的执行流程，不能随机地进入某一阶段，以保证正规检视的效果。具体流程见图16-2。

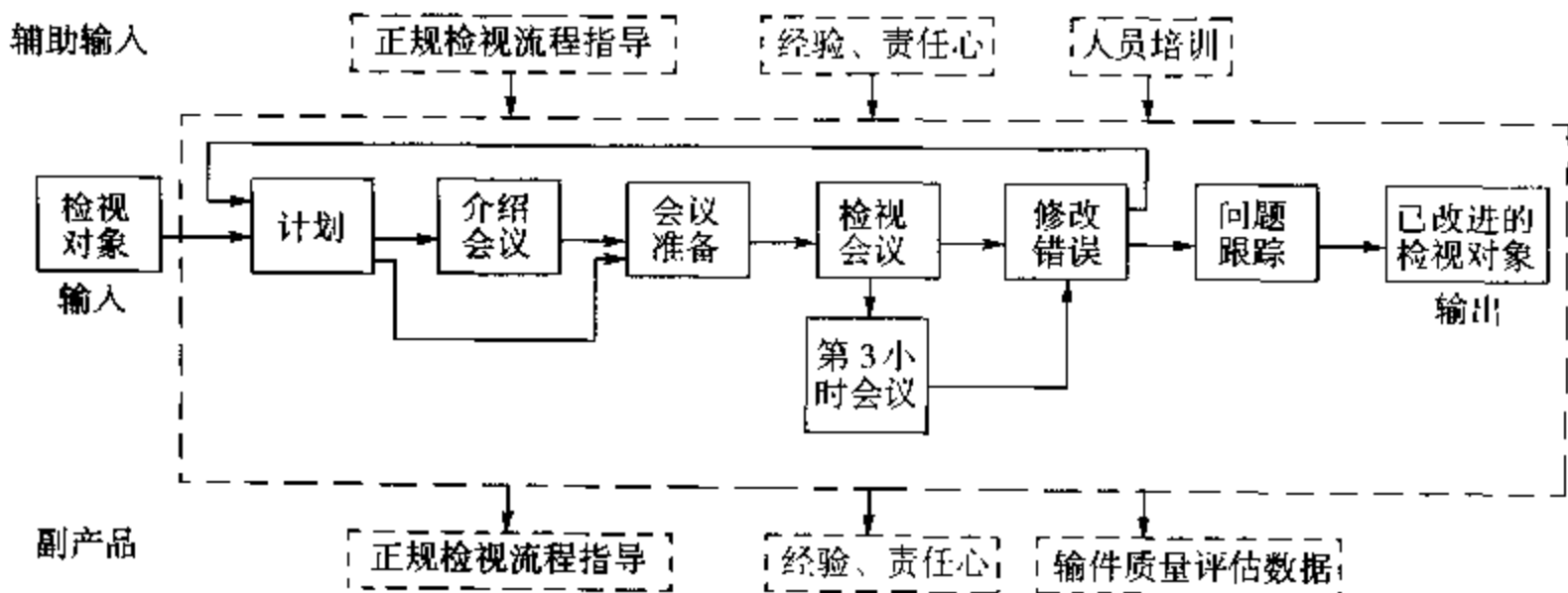


图 16-2 正规检视流程定义

1. 计划阶段

计划阶段工作主要由组织者完成，组织者需要完成的工作包括：保证检视对象已准备好，确定正规检视小组的成员和角色，确定会议的时间和地点，分发正规检视资料。正规检视的资料包括检视对象、正规检视通知单、正规检视问题登记表、相关资料和查检表。

组织者要确定被检视产品的内容量是否合理，保证正规检视工作能够在一次检视会议内完成。如果不能在一次检视会议内完成检视工作，组织者和开发者可把产品分解为几个易于处理的部分，分别进行正规检视。然后，组织者选择正规检视小组的成员并分配相应的任务，把正规检视资料袋分发给每一个检视小组成员。

组织者根据检视小组的成员对被检视产品资料的熟悉程度决定是否举行产品介绍会议。正规检视小组不仅需要知道开发产品的背景资料，还需要了解所开发完整系统的相关材料。在大部分情况下，正规检视小组的成员在项目组中都是业务骨干。如果检视小组不熟悉被检视的产品，就要安排召开产品介绍会议。

在正规检视的计划阶段，如果有开发项目管理数据库，就可通过已完成的检视项目的记录对组织者提供帮助。一旦组织者成立了正规检视小组，数据库可给每个检视者提供一个正规检视资料包。这个项目数据库可以提供参考资料和检视小组的资料备份。如果没有这种项目数据库或者项目数据库不能提供所需要的支持，这些工作必须由组织者完成。

本阶段的输入：

- 正规检视的对象；
- 与本次正规检视相关的资料。

本阶段的输出：

- 通知单；

- 资料袋。

本阶段资料袋的内容:

- 正规检视通知单;
- 正规检视对象;
- 参考资料, 可包括技术文档、标准和准则、以前正规检视的经验报告等;
- 问题登记表;
- 查检表。

本阶段开始准则:

- 已组建了由受过正规检视培训的人员组成的正规检视小组;
- 相关技术资料已由开发者提供给组织者;
- 其他条件已满足。如参加人员的时间安排、资料的熟悉程度等问题。

本阶段结束准则:

- 已确定了产品介绍会议的时间、地点(如需要介绍会议时);
- 已确定了检视会议的时间、地点;
- 每个检视小组成员的职责已分配;
- 资料袋已发放;
- 整个过程已记录。

是否需要产品介绍会议的准则:

- 检视对象的技术含量较高;
- 本检视对象是作第一次正规检视;
- 对检视者来说, 本次正规检视是第一次。

本阶段参加人员:

- 组织者;
- 开发者;
- 归档员。

2. 介绍会议

介绍会议阶段是可选择的阶段。如果检视小组成员不熟悉检视对象以及相关的背景, 那么这个会议就应当安排举行。在介绍会议上, 作者介绍产品的理论基础, 产品同被开发的系统其余部分的关系、产品的功能和产品的主要应用, 以及在产品开发过程中采用的开发方式。这些信息对检视小组是很必要的, 能够帮助检视小组成功地完成正规检视的工作。同时对产品非常感兴趣的非正规检视人员也可以参加介绍会议, 所有的检视者必须参加介绍会议。

召开介绍会议的主要原因如下:

- 正规检视小组不熟悉检视对象;
- 检视对象是新开发的或者是首次进行正规检视;
- 正规检视工作在检视对象的项目中被首次采用;
- 检视对象中应用了最新的技术。

本阶段需遵循的准则:

- 介绍会议的时间长度以 30 ~ 60 分钟为宜;

- 资料袋下发到召开介绍会议之前的准备时间间隔应不少于5个小时；
- 所有的检视者都必须出席产品介绍会议；
- 非工作人员也可以出席产品介绍会议；
- 开发者说而出席会议者只是听，不要进行细节讨论。

本阶段开始准则：

- 开发者已准备好；
- 检视者都已通知到。

本阶段结束准则：

- 会议结束；
- 主要检视者都参加了介绍会议；
- 整个过程已记录。

本阶段参加人员：

- 组织者；
- 开发者；
- 检视者。

3. 会议准备

会议准备阶段是正规检视过程的关键部分。在这个阶段中正规检视小组成员按照自己的角色单独准备检视会议。每个检视者都要仔细检查产品，检视者根据自己经验寻找产品中存在的普遍问题。在这个阶段，查检表用于引导发现检视对象中的典型错误。另外，检视者需保证检视对象同上游产品、标准规范和接口文档的一致性和正确性。检视者在对象审查过程中，要在问题登记表上记录发现的错误和审查过程花费的时间。在检视会议开始之前，将问题登记表提交给组织者。

正规检视会议开始之前，组织者检查每个检视者提交的问题登记表，判断正规检视小组是否进行了充分的准备。在正规检视期间，组织者需要特别注意检视对象中的难点，对错误能够很快进行分类。如果问题登记表显示检视小组准备不够充分，组织者需要调整检视会议时间，因为正规检视小组的准备工作不充分将会浪费时间并且不能够有效地发现错误。在正规检视会议上，问题登记表要返还给每个检视者，以便他们指出问题时使用。

本阶段的输入：

资料袋

本阶段的输出：

已填写完成的问题登记表

本阶段需遵循的准则：

- 检查产品错误所占用的时间应大于检视会议的时间；
- 检视者花费在检查产品上的时间一般为3~5个小时；
- 检视者应在检视会议召开4个小时之前将问题登记表提交给组织者，便于组织者组织一个更有效的会议和确定是否可以举行检视会议。

本阶段开始准则：

- 组织者已将资料袋发放到检视者手中；
- 产品介绍会议已召开(如需要)。

本阶段结束准则：

- 每一个检视者花费了足够的时间；
- 每一个检视者知道自己的职责；
- 每一个检视者已向组织者提交了填好的问题登记表；
- 每一个检视者已知道了检视会议的举行时间、地点；
- 整个过程已记录。

本阶段参加人员：

- 组织者；
- 检视者。

4. 检视会议

在检视会议期间，在讲解员对检视对象的详细解释下，检视者对检视对象一起进行审查，在需要时开发者可提供简短的解释，发现的错误都要被分类和记录。检视会议的焦点是发现错误。

组织者主持检视会议。如果正规检视小组是新组建的或者正规检视小组包含新的成员，组织者在检视会议开始时需介绍正规检视小组成员，简短描述成员的职责，重申正规检视的目的和检视对象。

讲解员开始按逻辑顺序解释检视对象，讲解时条理要清晰。解释内容包括：检视对象具体条目的功能，同相关上游文档的关系等。在检视会议上，如果阅读的内容包含任何可能的错误，检视者可以随时打断阅读。如果错误的确认需要讨论，则阅读被临时终止。记录员把所发现的错误记录在正规检视错误列表上，并进行相应的分类，阅读又重新开始。组织者应尽力限制讨论占用过多的检视会议时间。讨论时间的限制是非常重要的，如果讨论在规定的时间内没有结束，组织者将要声明此问题是无法确认的，并且检视会议继续进行，记录员也将记录这个无法确认的问题，留在第3个小时会议上解决。

关于每个错误是否是一个真正错误，检视小组的意见要达到一致。有时一个错误可能是检视者的观点错误，或者对开发者说明的理解错误。如果检视小组的意见不能够达成一致，这个错误就作为一个无法确认的错误被记录，留在第3小时会议上解决。这样就能够保证每个检视者发现的每个错误被解决的权利。记录员在审查列表中记录已确认的错误，包括错误的简单描述、错误的分类以及错误的发现者。

一个错误是否是一个真正错误的问题，可以引用上游文档进行分析判断，这种方法可以被检视者采用。如果讨论的是在上游文档中存在一个潜在错误，这时这个错误只作为一个无法确认的问题记录，检视会议要继续进行。这种错误的解决方案（无论是检视对象中的错误还是在上游文档中的错误）都要在第3小时会议上完成。记录员将无法确认的问题记录在审查列表中。

为了确定对所发现错误进行修改的优先次序，检视小组和组织者按照错误的严重程度（严重和一般）或错误分类标准把错误进行分类。例如，把导致系统不能够满足系统需求的错误划分为严重错误，所有其他类型的错误划分为一般错误（例如：书写错误，与辅助标准不一致的错误等）。根据每次正规检视中收集的数据信息，可以按照如下标准进行错误分类：数据错误、需求不一致、标准不一致、逻辑错误、接口错误、性能和可读性等。如果使用正规检视错误列表，这些分类信息都记录在每个错误的类型字段中。

在检视会议结束时,所有确认的错误总数被统计出来,组织者/或正规检视小组确定是否需要重新进行检视。这时,组织者也需确定是否需要召开第3小时会议。如果需要召集第3小时会议,这时就要给每个检视者指定会议议题。

在正规检视会议期间正规检视小组必须关注发现错误,而不要关心其他方面问题,例如问题的解决方法等。组织者有责任控制和调整检视会议,避免由于检视会议的时间过长导致正规检视小组成员过于疲劳,在这种情况下容易疏忽产品中的错误。因此,检视会议要限制在两个小时之内结束。如果在检视会议上没有完成产品的正规检视,就必须再安排一次检视会议。

在检视会议结束之后,开发者和组织者一起估算修改错误需要的时间,并且安排正规检视的问题跟踪会议。检视小组给开发者提供一个错误列表的备份,在修改过程中作为参考。注意:

- 对于在检视对象中发现的错误,不用填写正式描述报告(DRs)和正式更改请求(CRs)。不用填写被检视产品的正式描述报告(DRs)和正式更改请求(CRs)的原因是进行正规检视应当在检视对象进入版本配置管理控制之前;在保证解决了所有发现的错误,正规检视和再次进行正规检视结束之后,检视对象才能处在版本配置管理控制状态。
- 对于发现上游文档中的任何错误,都需要填写正式描述报告(DRs)。

本阶段的输入:

- 资料袋;
- 空白审查列表;
- 已完成的问题登记表。

本阶段的输出:

- 审查列表;
- 详细审查报告;
- 与上游产品中的问题相应的问题更改请求报告或问题报告(如果需要);
- 有详细问题标记的检视对象。

本阶段需要遵循的准则:

- 只检视产品,不评估开发者的能力;
- 发现问题是本阶段的惟一目的;
- 需要修改文档、手册中引起异议的部分;
- 检视会议的时间以不超过2个小时为宜;
- 在即将召开检视会议之前,不允许替换检视者;
- 项目经理不能参加检视会议;
- 检视会议的准备阶段结束到召开检视小组会议之前的时间应不少于5个小时。

本阶段开始准则:

所有的检视者都已准备好

本阶段结束准则:

- 检视对象已全部被检视;
- 确定是否要对本检视对象重新做正规检视;
- 检视小组确认的问题是否已记录和分类;

- 估计出修改错误所需要的时间；
- 确定了无法确认(解决)的问题的处理途径、时间；
- 组织者已记录了整个阶段所占用的时间；
- 当发现上游产品中的问题时，指定一名检视人员负责编写更改请求报告或问题报告。

需要进行重新正规检视的准则：

- 满足重新正规检视的标准，如由于本次正规检视，使检视对象的修改量占总量的比率达到某一量值时；
- 发现了大量的主要错误；
- 当修改完所有主要错误后，对检视对象内容产生了大的影响；
- 主要问题集中在检视对象的核心部分；
- 由于需求的改动，导致检视对象发生了较大的变化。

本阶段参加人员：

- 组织者；
- 检视者；
- 开发者；
- 讲解员；
- 记录员。

5. 第3小时会议

第3小时会议主要用于进行专题讨论，或者解决在检视会议上没有确认的问题。召开第3小时会议的原因，可能是开发者希望讨论已发现错误的解决方案，或是在检视会议上遗留的一些潜在严重错误还没有最后被确认，例如检视对象上游文档中潜在的严重错误需要进行解决。第3小时会议可以召开一个小组会议，也可以由检视小组成员安排自己的时间，单独完成指定内容后汇报结果。

第3小时会议不必在检视会议结束之后马上开始，也没有必要严格限制在一个小时之内。如果采用小组会议的方式，第3小时会议可以讨论错误的解决方案和解决有分歧的错误。出席第3小时会议的人可以是正规检视小组的成员，也可以不是正规检视小组的成员，包括相关的管理人员(仅仅是由于技术方面原因参加会议)，外单位的技术专家，与错误相关的人和能够解决错误的人员。在很多情况下，只有那些有兴趣的检视者参加会议。第3小时会议能够提供给开发者解决错误的资料，便于开发者更有效地修正错误；可以报告所发现上游文档的严重错误；以及解决在检视会议上没有确认的所有错误。

如果第3小时会议是检视者单独进行检视，那么其主要目的是确认在检视会议上无法确认的问题，寻找解决此类错误的资料，如果上游文档处在配置管理状态之下，严重的错误要填写错误描述报告或者更改请求表。

本阶段的输入：

无法确认的问题

本阶段的输出：

- 处理无法确认问题的解决方案；

- 提供给开发者用于修改问题时可参考的资料；
- 修改问题的建议；
- 问题报告或更改请求报告；
- 一致同意现在无法(或不必要)解决的问题的记录。

本阶段需要遵循的准则：

- 是一个比较松散的阶段；
- 只在有要求时，才进入本阶段。

本阶段开始准则：

- 开发者的要求；
- 需要填写问题更改请求报告或问题报告；
- 需要检视者讨论无法确认问题的处理方法、途径。

本阶段结束准则：

- 向开发者提供了解决问题的方法、途径；
- 完成填写问题更改请求报告或问题报告；
- 现在无法(或不必要)解决的问题已记录；
- 组织者已记录了本阶段所花费的时间。

本阶段参加人员：

- 组织者；
- 检视者；
- 开发者；
- 其他人员(主要是与解决无法确认问题有关的人员)。

6. 修改错误

修改错误阶段主要是修改在正规检视过程中已确认的错误。开发者负责修改审查列表中记录的所有严重错误。在时间和条件允许的情况下，也要修改一般错误。组织者要经常和开发者沟通，提供给开发者关于未解决错误的信息以及修改错误所需要资料。

本阶段的输入：

- 检视错误列表；
- 已标出问题的检视对象；
- 第3小时会议中提出的解决无法确认问题的方法、途径。

本阶段的输出：

- 检视对象中已发现的问题是否已全部修改；
- 整个阶段所花费的时间的记录。

本阶段需要遵循的准则：

- 处理所有主要的问题；
- 根据实际情况，需要考虑其他问题的处理方式；
- 需要5~20小时；
- 严格按照计划完成修改工作。

本阶段开始准则：

- 前一阶段工作已全部完成；
- 开发者已拿到检视列表；
- 开发者已拿到有问题标注的检视对象；
- 开发者已了解无法确认问题的有关情况。

本阶段结束准则：

- 开发者的修改工作已完成；
- 在检视列表上注明了已修改的问题；
- 与组织者确定了召开问题跟踪会议的时间。

本阶段参加人员：

开发者。

7. 问题跟踪

问题跟踪阶段是组织者和开发者之间的简短会议，保证在正规检视期间已确认的所有严重错误都被改正，并且没有由于修改而产生新的错误。开发者和组织者要斟酌每个错误的修改过程和方案。如果一般错误也修改完成，那么也要检查一般错误的修改，但不是重点。组织者要保证满足正规检视的结束标准。如果需要特殊的专业技术，组织者可要求其他的技术专家参加本阶段工作。

8. 重新正规检视

如果在检视过程中发现非常多的错误或者错误改正非常复杂，应当对此检视对象再次进行正规检视。是否需要再次进行正规检视在检视会议结束之后由组织者和开发者决定。如果发现的所有严重错误都被改正，所有无法确认的问题已经解决，并且产品已经满足正规检视的结束标准，那么组织者就可以批准检视对象通过正规检视，并在正规检视的报告中记录完整的正规检视过程。如果那些条件还没有满足，还需要返回到错误修改阶段，开发者要进行必要的修改。

本阶段的输入：

- 错误已经改正的检视对象；
- 开发者已填写的检视表。

本阶段的输出：

- 通过了正规检视的检视对象；
- 组织者已确认的检视列表；
- 组织者已妥善处理了无法确认问题，完成了问题报告和更改请求报告的编写；
- 检视综合报告；
- 需要归档的资料袋。

本阶段需要遵循的准则：

- 整个过程(会议)不要超过 2 个小时；
- 确保所有主要错误已正确修改；
- 确保所有无法确认问题已妥善处理。

本阶段开始准则:

- 开发者已完成了错误修改工作;
- 所有的无法确认问题已妥善处理。

本阶段结束准则:

- 对检视对象的正规检视已圆满完成;
- 详细检视报告已提交给正规检视数据管理者;
- 资料袋已归档;
- 检视综合报告已分发给开发者、归档员、质量管理者、正规检视数据管理者;
- 正规检视的结果已发给检视小组成员和项目经理;
- 问题报告和更改请求报告已归档。

本阶段的资料袋中所包括的内容:

- 已通过正规检视的检视对象;
- 正规检视通知单;
- 相关资料的复印件;
- 问题登记表;
- 查检表;
- 详细审查报告;
- 检视综合报告;
- 问题报告、更改请求报告、现阶段无法(或不必要)解决问题登记表的复印件;
- 参考文献(或参考文献列表);
- 更有效的查检表。

本阶段参加人员:

- 开发者;
- 组织者;
- 归档员。

16.5.3 正规检视常用表格

在正规检视过程中,经常使用到各种表格,包括:

- 正规检视通知单(参考表 16-3),由组织者完成,在正规检视过程开始的时候发给各参与者;
- 正规检视问题登记表(参考表 16-4),由各检视者在会议准备期间完成,在正规检视会议开始前4个小时提交给组织者;
- 正规检视审查列表(参考表 16-5),由记录员在检视会议期间完成,并且开发人员在检视会议后根据该审查列表修改错误;
- 正规检视详细报告(参考表 16-6);
- 正规检视综合报告(参考表 16-7),正规检视详细报告和综合报告是在正规检视过程结束的时候由组织者完成的,最终提交给管理者审查。

表 16-3 正规检视通知单

填发日期:

序列号:

正 规 检 视 通 知 单					
检视对象					
介绍会议	日期		检视会议	日期	
	时间			时间	
	地点			地点	
是否重新正规检视		<input type="checkbox"/> 否 <input type="checkbox"/> 是 原因: _____			
正规检视 小组成员	姓 名	部 门 / 电 话	职 务	技术职称	
			组织者		
			记录员		
			讲解员		
			审查者		
			审查者		
			审查者		
			审查者		
检视对象类型: <input type="checkbox"/> 系统需求 <input type="checkbox"/> 子系统需求 <input type="checkbox"/> 功能设计 <input type="checkbox"/> 软件需求 <input type="checkbox"/> 概要设计 <input type="checkbox"/> 详细设计 <input type="checkbox"/> 程序代码 <input type="checkbox"/> 测试计划 <input type="checkbox"/> 测试用例 <input type="checkbox"/> 测试过程设计 <input type="checkbox"/> 操作手册 <input type="checkbox"/> 规范、标准 <input type="checkbox"/> 其他: _____					
检视对象的性质: 新内容: _____ % 修改内容: _____ % 重用内容: _____ % 规模(范围): _____					
参考资料	名称				作者
备注					
填表人		部门		电话	

表 16-4 正规检视问题登记表

填表人：

序列号：

正规检视问题登记表				
审查对象：_____				
开始时间：_____ 结束时间：_____				
审查记录	审查日期		花费时间	
问题列表：（将在正规检视过程中发现的问题详细记录在下面的表格中）				
序号		出处	错误类型	
描述及分析			<input type="checkbox"/> 错误 <input type="checkbox"/> 严重 <input type="checkbox"/> 建议 <input type="checkbox"/> 一般 <input type="checkbox"/> 其他 <input type="checkbox"/> 规范 <input type="checkbox"/> 未确认 <input type="checkbox"/> 确认 发现日期：_____	
序号		出处	错误类型	
描述及分析			<input type="checkbox"/> 错误 <input type="checkbox"/> 严重 <input type="checkbox"/> 建议 <input type="checkbox"/> 一般 <input type="checkbox"/> 其他 <input type="checkbox"/> 规范 <input type="checkbox"/> 未确认 <input type="checkbox"/> 确认 发现日期：_____	
序号		出处	错误类型	
描述及分析			<input type="checkbox"/> 错误 <input type="checkbox"/> 严重 <input type="checkbox"/> 建议 <input type="checkbox"/> 一般 <input type="checkbox"/> 其他 <input type="checkbox"/> 规范 <input type="checkbox"/> 未确认 <input type="checkbox"/> 确认 发现日期：_____	
序号		出处	错误类型	
描述及分析			<input type="checkbox"/> 错误 <input type="checkbox"/> 严重 <input type="checkbox"/> 建议 <input type="checkbox"/> 一般 <input type="checkbox"/> 其他 <input type="checkbox"/> 规范 <input type="checkbox"/> 未确认 <input type="checkbox"/> 确认 发现日期：_____	
<input type="checkbox"/> 同意召开小组审查会议 <input type="checkbox"/> 不同意召开小组审查会议，原因：_____				
注：必须在召开审查小组会议 4 个小时之前将此表提交给联络员				

表 16-5 正规检视审查列表

填表人/日期: _____

序列号: _____

正规检视审查列表					
正规检视对象: _____ 修改错误所需时间(估计): _____ 问题跟踪会议时间: _____ 错误修改完成日期(估计): _____ 错误修改实际完成日期: _____					
序号	出处	错误类型	备注	发现者	
描述分析		<input type="checkbox"/> 错误 <input type="checkbox"/> 严重 <input type="checkbox"/> 建议 <input type="checkbox"/> 一般 <input type="checkbox"/> 其他 <input type="checkbox"/> 规范 类型:			
修改说明: _____ <input type="checkbox"/> 已正确修改 <input type="checkbox"/> 没有正确修改, 原因: _____					
序号	出处	错误类型	备注	发现者	
描述分析		<input type="checkbox"/> 错误 <input type="checkbox"/> 严重 <input type="checkbox"/> 建议 <input type="checkbox"/> 一般 <input type="checkbox"/> 其他 <input type="checkbox"/> 规范 类型:			
修改说明: _____ <input type="checkbox"/> 已正确修改 <input type="checkbox"/> 没有正确修改, 原因: _____					
序号	出处	错误类型	备注	发现者	
描述分析		<input type="checkbox"/> 错误 <input type="checkbox"/> 严重 <input type="checkbox"/> 建议 <input type="checkbox"/> 一般 <input type="checkbox"/> 其他 <input type="checkbox"/> 规范 类型:			
修改说明: _____ <input type="checkbox"/> 已正确修改 <input type="checkbox"/> 没有正确修改, 原因: _____					
序号	出处	错误类型	备注	发现者	
描述分析		<input type="checkbox"/> 错误 <input type="checkbox"/> 严重 <input type="checkbox"/> 建议 <input type="checkbox"/> 一般 <input type="checkbox"/> 其他 <input type="checkbox"/> 规范 类型:			
修改说明: _____ <input type="checkbox"/> 已正确修改 <input type="checkbox"/> 没有正确修改, 原因: _____					

表 16-6 正规检视详细报告

完成日期:

序列号:

正规检视详细审查报告						
审查对象						
已发现问题统计						
错误种类 (查检表中的定义)	错误		建议	规范	其他	总计
	严重	轻度				
总计						
无法确认问题处理情况:						
序号	描 述					责任人
备注:						
此表填写完后请及时返回到数据管理员_____处 组织者签名:						

16.6 本章小结

本章介绍了同行评审的概念以及同行评审的3种形式——走读、技术评审和正规检视。尽管同行评审可以适用于任何工作产品，可以在开发阶段的任何一个时间点进行，但是还有一个成本的问题。不同的同行评审类型，根据其过程的严格程度，其成本是不同的。走读形式最自由，因此需要的成本也最低；其次是技术评审。正规检视是成本最高的，因此它应当被用到最值得付出的产品项上。一般正规检视在需求，设计文档上面应用的比较多一些，而走读可以应用到各种产品项上（包括文档和代码）。技术评审一般是在阶段点上进行，以确认某个阶段的工作已经完成，并且达到一定的技术要求，可以进入到下一个阶段。

根据经验，在开发过程中可以按照下面的方法安排各类同行评审：

方法一 在某个产品项（包括文档和代码）的开发过程中，可以进行多次走读，像在印度一些CMM5级的公司经常进行每日走读的方式，例如：在编码时，员工在一天8小时的工作中，把7个小时左右的时间放在编码上，再利用其余的1个小时进行交叉走读。这种方式是一个非常好的实践，能够有效地减少低级错误，并提高产品质量。

方法二 在某个产品项已经完成（包括文档和代码），完成的概念是指已经经过多次走读，并且可以准备提交进入基线了。这时，对于一些关键性文档或代码进行一次或多次的正规检视，次数的多少需要看检视对象的规模大小。这个过程如果组织得好，是非常高效的。

方法三 在一个开发阶段结束，并且目标里程碑中所包含的所有产品项（包括文档和代码）都已经完成，且都被基线化了。这时可以进行一个技术评审，该评审确认任务的完成情况和完成质量，并结束当前阶段，开始启动下一个阶段。

正规检视的次数安排不能太多，一般一个人一周最多只能参加1次正规检视，否则会导致人员疲惫，影响检视效率。同时检视的效率还与员工花在检视准备和执行上面的时间和工作量有关。例如，IBM发现当检视速率超过一定值时检视效率会成倍下降。适当的检视速率取决于被检视的产品的类型和参加检视人员的经验。对设计和代码检视来说，表16-8显示了一些可用的检视速率的数据，从中我们可以看到，COBOL应用程序的检视速率是一般系统程序检视速率的6~8倍，是性能敏感系统程序的15倍。组织确定自己检视速率的最好的方式是收集自己的数据并得出适合自己的标准值。

表 16-8 检视速率数据

类型	详细设计		编码	
	预审	评审	预审	评审
IBM COBOL	898	652	709	539
IBM System	100	130	125	150
IBM System			125	90
Bell Labs System				100
Bell Labs, microcode				

随着组织经验的积累,同行评审(尤其是正规检视)效率也越来越高。通过提高检视效率,组织能够在软件交付前发现大部分的错误(表 16-9 给出了一个 IBM 统计的实例)。另一方面,检视所花费的时间也有很明显的变化:如 1000 行源代码所花费的时间增加了 1.68 倍(概要设计检视)和 7.00 倍(详细设计检视)。尽管在前期设计检视上面花费了额外的时间,但在代码检视阶段每千行源代码的检视时间却减少到原来的 76%,而每小时检视发现的问题数更是增加到原来的 3 倍。这些改进一方面是由于产品程序员们对产品的了解更加深入所致,另一方面则是因为检视是一种可以学习的技能,可以通过经验的积累而得到明显的改善。

表 16-9 组织改进数据

版本	Relative * Hours/KCSI			Relative * Hours/Defect			检视 效率 *	质量 索引 *
	I0	I1	I2	I0	I1	I2		
R1	1.00	1.00	1.00	1.00	1.00	1.00	0.15	100%
R2	1.27	8.16	0.99	2.79	1.00	0.46	0.41	55%
R3	1.68	7.00	0.76	0.18	0.53	0.33	0.61	25%

其中:KCSI:千行变更的源指令

I0:概要设计检视

I1:详细设计检视

I2:代码检视

检视效率 = $\frac{\text{检视发现缺陷数}}{\text{检视和测试发现的总缺陷数}}$

质量索引 = $\frac{\text{试用期间发现的缺陷数}}{\text{开发期间发现的总缺陷数}}$

第 17 章 测试经验总结

正如第 1 章所说那样，测试是一个极其广泛的领域，需要花费大量的时间和精力去研究和实践。尽管本书尽可能详细地介绍了测试的方方面面，然而这也只能包含测试的一个方面。还有很多测试方面的知识和概念无法一一进行描述。正如同其他知识领域一样，不可能在完全掌握了所有知识之后才开始应用。测试的知识是在不断的实践中积累并发展的。本章将谈论测试在实际应用中需要注意的一些事项，包括一些原则、实践经验等内容。通过本章的学习，你将了解以下内容：

1. 为什么说测试是一个过程而不是一个阶段？
2. 一个好的测试计划具有哪些特征？
3. 测试为什么不可以穷尽？
4. 应当如何看待测试和开发的关系？
5. 如何看待测试自动化的作用？
6. 为什么要尽早地、频率地测试？
7. 测试度量包含哪些数据内容？
8. 记住软件测试的 10 大原则和 10 大实践。

17.1 软件测试的 10 大原则

17.1.1 原则 1：测试是一个持续进行的过程，而不是一个阶段

在传统的瀑布式开发模型中定义了专门的测试阶段，如单元测试阶段，集成测试阶段或系统测试阶段。然而，这并不意味着测试只有在这个时候才进行。笔者遇到过很多项目，在这些项目中，对测试的理解都基于了阶段这个概念，在他们的思维中，测试只有在适当的时候才开始，并且在某个点就可以结束。这是一个错误的理解，并且对产品的质量来说是很危险的。其实在本书中我们不仅一次地强调，现代的测试已经发展成为一个全过程的验证和确认活动，它贯穿于整个开发生命周期的始末^{[3][8][18]}。为了获得最大的受益，测试的开发和准备必须在编码之前就开始，同时为了保证最终的质量，必须在开发过程的每个阶段都保证其过程的质量。

17.1.2 原则 2：测试必须被计划、被控制，并且被提供时间和资源

测试并不是一个随机的活动，测试必须被计划，并且被安排足够的时间和资

源^{[8][26][263][264]}。测试活动应当受到控制，测试的中间产物应当被评审并纳入配置管理^{[263][264]}。

测试计划是一个关键的管理功能，它定义了各个级别的测试所使用的策略、方法、测试环境、测试通过或失败准则等内容。测试计划的目的是要为有组织地完成测试提供一个基础。从管理的角度来看，测试计划是最重要的文档，这是由于它帮助管理测试项目。如果一个测试计划是完整并且经过深思熟虑的，那么测试的执行和分析将平滑地进行^[8]。

测试计划可以分级，也可以是一个总的计划，并且测试计划是一个不断演进的文档。如果不考虑应用软件的最初来源（复用的组件或已实现的组件），软件需求是测试活动的驱动。因此，测试计划应当关注于文档化的需求。此外，支持测试的过程应当被文档化下来以创建一个可重复的过程，该过程将保证开发工作产品的质量。

一个好的测试计划应当：

- 在检测主要缺陷方面有一个好的选择；
- 提供绝大部分代码的覆盖率；
- 是灵活的；
- 易于执行、回归和自动化；
- 定义要执行测试的种类；
- 清晰地文档化了期望的结果；
- 当缺陷被发现时，提供缺陷核对；
- 清晰地定义测试的目标；
- 明确测试的策略；
- 清晰定义测试的出口标准；
- 没有冗余；
- 确认风险；
- 文档化测试的需求；
- 定义可交付的测试件。

17.1.3 原则3：测试应当分级别

在我们的开发过程中，系统的设计和实现是有一定的层次的，从原始需求到高层的构架设计，再到详细设计，最后到编码。同样，针对这种开发的层次，测试也应当有一定的层次级别，类似我们前面所说的单元测试、集成测试、系统测试、验收测试等。我们不当跨越底层的测试级别（例如单元测试）而直接进入较高的测试级别（例如系统测试）。具体的原因在前面的章节中已经详细讲过，这里就不再赘述。当然在某些特殊情况下，这是允许的，例如对于很小的系统。

测试的级别定义有多种方式，如果从全过程测试来看，还可以扩展到同行评审类型上。Norm Brown 把测试级别分成了9个层次（0~8），具体参考表17-1^[264]。

表 17-1 测试级别

测试级别	测试活动	测试类型	测试的文档基础	测试责任主体	测试关注点
级别 0	结构化检视	非计算机测试 (静态测试)	各类文档	检视组	各方面
级别 1	计算机软件 单元测试	白盒测试	软件详细设计 文档	开发人员	软件单元设计
级别 2	计算机软件 配置项集成 测试	白盒测试	软件概要设计 文档	独立测试组	计算机软件配 置项设计/构架
级别 3	计算机软件 配置项资格 测试	黑盒测试	软件需求规格 说明书	独立测试组	计算机软件配 置项需求
级别 4	计算机软/ 硬件配置项 集成测试	白盒测试	系统的子系统 设计文档	独立测试组	系统设计/构架
级别 5	系统测试	黑盒测试	系统规格说明书	独立测试组	系统需求
级别 6	DT&E 测试	黑盒测试	用户手册	独立测试组	用户手册的一 致性
级别 7	OT&E 测试	黑盒测试	可操作性需求 文档	可操作性测试组	可操作性需求
级别 8	外场测试	黑盒测试	交付计划(场地 配置)	外场安装组	场地需求

其中: DT&E——Development Test and Evaluation

OT&E——Operational Test and Evaluation

17.1.4 原则 4: 测试应当有重点

尽管测试需要按照一定的级别进行,但资源和时间是有限的,实际上我们不可能无休止地进行测试,因此在有限的时间和资源下如何有重点地进行测试是测试管理者需要充分考虑的事情。例如,在单元测试时,对于哪些函数需要重点测试,哪些函数可以粗略测试,哪些函数可以不测试;而对于系统测试,则要考虑首先应当保证哪些功能的测试,其次应当保证哪些功能的测试等。测试的重点选择需要根据多个方面考虑,包括测试对象的关键程度,可能的风险,质量要求等。这些考虑与经验有关,随着实践经验的增长,判断也会更有效。

17.1.5 原则 5: 测试不是为了证明程序的正确性,而是为了证明程序不能工作

正如 Mayer 所说,测试的目的是证伪而不是证真^[2]。事实上,证明程序的正确性是不可能的,一个大型的集成化的软件系统不能被穷尽测试以遍历其每条路径。我们前面也分析过,即使遍历了所有的路径,错误仍有可能隐藏^{[2][266]}。我们做测试是为了尽可能地发

现错误。因此，测试必须包含一系列测试级别，就如表 16-1 所说的。这些测试级别最大化了测试的覆盖率。

必须有一些标准可以用于平均所有的测试活动。所有可以跟踪到需求的测试可以通过 3 个方式进行执行：

- 在正常的数据流量下的有效信息；
- 在一个控制环境中使用超量的数据输入速率；
- 使用一个预先计划的正常数据和异常数据的组合。

理想的测试环境要能够使得一个系统在可控的方式下被破坏。例如，数据及数据组合必须不断变化直到系统不能够以正常的方式接受。系统支持变得不可接受的点必须被确认并文档化下来。

必须在所有的测试级别上运行测试，且同时使用正常条件和异常条件。这是很严格的，即使在测试环境难以建立的情况下。

17.1.6 原则 6：测试是不可能穷尽的，当测试出口条件满足时就可以停止测试

我们说测试是为了发现错误，一个好的测试是发现以前没有发现的错误^[2]。但是这个要求可能会使人走入极端。其实，不同的系统有着不同的质量要求，对于质量要求严格的系统，可能需要进行长时间的、全面的测试，尽可能地挖掘系统中的缺陷。然而对于质量要求不是很严格的系统，系统是允许出现错误的，因此我们通过测试的目的是使系统的缺陷数量降到可接受的范围内。

类似原则 5 我们已经提到测试是不可能穷尽的，原则 4 也说到资源和时间是有限的。因此在做测试时需要分析哪些功能是对用户很关键，在这些功能中出现某类型错误对用户是不可接受的，而相对其他一些功能，出现的错误是可以容忍的，这样，我们在测试时，重点就应当去寻找那些用户不可接受的错误，而不是漫无目的地去搜索错误。同时我们应当对测试定义合理的出口标准，这是因为测试是没有穷尽的，终归会发现系统的问题，然而我们不能无休止地去找寻这些问题。当条件满足时，我们就应当停止测试。而测试出口条件的设置需要考虑系统的质量要求及系统的资源要求。曾经有人说过：当时间和资源用尽时，测试也就停止了。这是没有办法的最好办法。

17.1.7 原则 7：测试是开发的朋友，不是开发的敌人

测试人员和开发人员经常无法有效地一起工作。这一方面是因为双方工作的性质不同（开发的工作是构建系统，而测试的工作是去破坏系统），另一方面也可能是因为管理的原因造成了测试和开发之间的矛盾。不管是什么原因，这个矛盾，对于产品来说不是一件好事。

如何处理好测试与开发之间的关系是现代软件管理研究中的一个课题。开发和测试作为一个整体都是服务于产品，都要为产品的质量负责。从这一点来讲，开发和测试的利益是一致的。要知道，如果在产品交付使用之前，测试人员遗漏了一个问题，而这个问题最

终在用户手上被发现，并产生比较严重的后果时，那么，无论是开发还是测试，最终都逃避不了责任。我们要为质量服务，测试的目的是要去找错误，最终提高产品的质量，而不是去找开发的花，只有当双方都认识到这一点时，开发和测试就有共同交流的基础了。测试人员应当是开发人员的朋友，他们帮助开发者寻找遗留在产品中的缺陷，使得开发人员能够产生更好的产品。测试和开发不应当是敌人。

尽管有很多开发人员和测试人员也认识到了这一点，然而，在实际操作上，总不免产生摩擦，Brian Marick 在如何处理这方面矛盾上给出了一些好的建议^[265]：

- 以一种开发人员希望你在周围的方式定义自己的角色。帮助开发人员在缺陷被别人看到之前排除掉缺陷，这可以降低整个项目的成本；
- 解释自己和自己的工作，以便在疑惑的时候可以获得开发人员的帮助。降低开发人员对你进行不利判断的概率，并采取一些方式让开发人员相信缺陷报告不应被看成是一种威胁；
- 缺陷报告的书写应当清晰、无歧义。

17.1.8 原则8：测试人员应公正地测试，如实地记录和报告缺陷

我们说测试是开发的朋友，这并不等于测试就应当处处维护开发人员，替开发人员隐瞒缺陷或纵容缺陷的存在。这是对朋友的误解。我们要知道缺陷的存在最终只会影响到整个产品。在开发过程中，测试人员一直承担着质量把关人员的角色，尤其是测试将会是产品最终交付给用户之前的最后一道关卡。如果测试人员不能站在公正的立场上去执行测试，并如实地记录和报告缺陷，那么最后受伤的不仅仅是开发人员，还会包括测试人员自己。对质量负责，不仅仅是对自己负责，也是对开发负责和对产品负责。

17.1.9 原则9：测试自动化能解决一部分问题，但不是全部

工具所能发挥的作用依赖于使用工具的人。因此，对工具的过分依赖将降低人的能动性，并最终使测试本身受到损害。适当地使用测试工具能够减轻测试人员的机械性工作，提高工作效率，而滥用工具会降低测试的质量。并不是任何工作都适合自动化的，如何合理地自动化测试，合理地选择适当的测试工具已经成为研究人员感兴趣的一个课题。Mark Fewster 和 Dorothy Graham 在这方面做了很多工作^[17]。Brian Marick 和 Cem Kaner 也提出了很多实践性的建议^{[267][268]}。笔者即将出版的《软件测试自动化》也在这方面做了一些探索和总结。

17.1.10 原则10：测试不能仅仅包括功能性的验证，还应当包含性能、可靠性、可维护性、安全性等方面的验证

笔者遇到过一些产品的测试，其范围仅局限在功能领域内进行测试。这一方面可能有产品进度的压力影响，另一方面则是测试人员对测试的理解还比较局限。从用户角度来讲，其需求除了功能性需求外，还包括非功能性需求，有些非功能需求可能是显性的，而

有些非功能需求则是隐性的。我们在测试时，应当关注所有的需求，在验证功能的同时，还需要验证产品的性能、可靠性、稳定性、可维护性、安全性、可操作性、可安装性等。一个产品的缺陷往往会在其性能的边界上产生，如果我们忽视了这部分的测试，很多缺陷将漏过测试进入到用户手上。

17.2 软件测试的 10 个最佳实践

17.2.1 实践 1：尽早地、频繁地进行测试是降低项目成本，提高质量的一个好方法

现代测试的一个重要哲学要求尽可能早地，尽可能频繁地进行测试，尽可能多地从开发处获得反馈信息^[3]。这包含要求测试尽可能早地进行准备，并且和开发人员一起进行评审、走读、单元测试、原型评价、早期模拟等。早期测试的目的是尽可能早地发现任何意想不到的坏的消息，并且帮助开发人员产生高质量的单元。

该方法希望在缺陷产生时发现并纠正缺陷，它假设了在早期测试中发现的问题能够被描述并及时修正。许多项目管理人员延迟了缺陷修正的时间直到开发人员已经完成了所有特性的设计和编码。这大大提高了系统出错的可能，也增加了修改的成本。一般来说，一次完成一个特性的设计和编码，并保证其正确性将更加有效。

为什么要尽早地发现缺陷和修正缺陷呢？这主要有以下原因：

- 正如第 1 章所描述的一样，缺陷的修改成本随着阶段的推移将急剧上升，在产品发布之后修正一个缺陷的成本将是需求阶段的 100 倍，甚至更高（参考图 1-1）；
- 缺陷具有放大的特点，缺陷修改延迟几个星期甚至几个月将使得系统更容易出错；
- 设计判定和一些小的代码限制及条件很容易被忘掉；
- 尽早修正缺陷可以节省重新分析设计的时间；
- 早期的问题反馈有助于防止类似错误的产生；
- 大量的缺陷和问题跟踪工作将被减轻；
- 如果必要的话，可以重新设计和编码，而这个工作越往后期越不可能。

17.2.2 实践 2：尽早产生一个综合的主测试计划

提供和维护一个主测试计划（Master Testing Plan），包含所有预期的测试活动和测试交付物。综合的测试计划应当结合总的项目和程序开发计划，并保证资源和责任在项目中尽可能早地被了解和分配。

大部分项目没有尽早地描述测试的问题。主测试计划解决了这个问题并且使得测试的工作和策略可以被项目中的每个人看到和理解。主测试计划至少应当包括测试的总工作量，分配所有主要工作的责任以及所有测试级别上应交付的物件。其目的是要提供一个大的活动图并且协调所有的测试工作。

主测试计划涉及到项目组所有成员,包括用户、客户和管理人员。在测试评价中所包含的每个人的活动应当被描述,包括那些分配给开发人员的活动,例如:单元评审和测试。产品经理以及那些在项目之外的人员将发现主测试计划有助于把测试过程融合到整个项目开发过程中去。随着项目的进行,主测试计划也将被修正和更新。

创建主测试计划不需要花费许多工作量,并且它不需要一个特别冗长的或者严肃的文档。许多内容可以通过类似于RAD,头脑风暴等方法在项目早期被完成。

17.2.3 实践3:对质量要求较高或大型复杂的产品成立独立的测试组

测试是一个需要专业技能的工作,它需要专门的培训和实践,不应当把它看作是一个开发之外附带的工作。大部分人认为(尤其是项目管理人员)测试是项目工作中的一部分,因此他们认为只要能够小心地彻底地测试自己的工作,就能够提交高质量的工作结果。我们完全可以理解好的测试是每个人的任务,也知道测试是过程的一部分,并且在大部分情况下,我们知道需要做什么,需要得到什么。然而,事实上在进度和竞争的压力下,测试任务经常被第一个延迟、裁减或完全绕开。为什么会这样?我们应怎样做才能使组内每个人更好地履行他的责任?

大部分问题是心理上的问题:

- 当我们知道会有更多的问题被发现时,测试会让人沮丧;
- 当我们认为每一样东西都已经被完成时,测试让人厌烦;
- 弱的测试让我们感觉得不偿失;
- 对开发人员来说,测试暴露的问题越多,意味着有更多的返工需要进行;
- 对管理人员来说,测试看起来像是虚的工作,在项目陷入问题时是可以被砍掉的;
- 开发人员相信用户和客户愿意购买低质量的,但功能更多的产品。

由一个独立于开发的专业测试组来从事项目的测试是一个比较好的实践,并且能有效解决上面的问题。

17.2.4 实践4:在每个开发阶段,使用测试和评价的结果作为是否可以通过的标准

许多项目看上去都进行得非常不错,直到它们进入集成测试和系统测试,这时,甚至一个简单的测试都无法运行起来。如何避免这种现象的产生?一个好的方法是尽早测试,并且在每个阶段上通过测试数据来评价其是否能通过阶段的质量标准。当你坚持使用测试来证明每一个特性或对象被完成,那么就能保证所有的设计和代码被真正地完成。

17.2.5 实践5:开发和维护一个测试需求和目标的风险优先级列表

现代测试的一个重要实践是在测试被设计和创建之前建立一个需要覆盖的目标清单。这个清单细化了需要测试的内容,并且被用于指导和驱动测试设计和开发工作。根据风险

的优先级(一般分为高、中、低)有助于使测试关注于高风险区域。

下面是一个如何获取测试清单的指导:

- 尽可能早地开始并且覆盖所有需求或功能设计规格;
- 使用头脑风暴和非正式会议来创建一些大家担心的事项列表;
- 在评审期间,评价清单中的每一项,并设定优先级;
- 把清单中的内容分成三个主要的增量组:需求、设计和代码;
- 把每个增量组分解成一个小部分的逻辑组。

由于清单列表可能会变得非常长,因此把它们分类是很有帮助的。遗漏的目标可以很容易从这些组中被确认。

随着每个测试被开发,相关的清单目标和组被进行了记录,作为测试描述的一部分。通过一些简单的测试管理工具,可以报告需求和目标的测试覆盖情况,尤其是那些还没有被任何测试覆盖到的目标。

17.2.6 实践6:把测试件作为产品的一部分等同管理,使用相同的评价标准和过程

这里,测试件包括了开发的测试工具、测试套、测试驱动程序、测试桩模块等。认识到测试件也是软件,也需要像其他软件一样被管理和被工程化,这一点很重要。事实上,测试件是一个特殊的应用系统,它的主要目的是为了测试和评价别的应用或系统。如果应用是关键,那么该应用的测试件应当也是关键的,并且必须使用好的工程原则,包括适当的测试和评价(针对测试件的测试)。

对于工程化测试件,其生命周期过程是和开发的软件完全一样的。一些好的软件工程概念和原则都可以被应用到测试件的开发上。有效的测试件工程化关键是获得合适的时间。如果你创建测试件太迟,并且在大部分软件组件已经被设计和编码之后,那么你就不能获得预想的好处和积极的反馈。如果你创建得太早,在软件设计和需求稳定之前就完成,那么就不能够有效地开发出符合测试需求和目标的测试件,并且你需要面临着很多的返工。

17.2.7 实践7:提供集成化的测试工具和测试基础支持

为了更好地测试,我们需要有一个合适的测试自动化平台和一个组织良好的测试数据库。正如在原则9所提到的一样,自动化能解决一部分问题,但不能解决所有问题。但是没有自动化的测试将会是痛苦的,尤其在一些大系统上进行的专项性测试。例如,我们所说的负载测试、容量测试、可靠性测试等。这类测试,如果没有工具的支持是很难完成的。一个好的测试自动化实践是根据项目的需要,综合选择多种自动化应用平台,包括:

- 测试资产跟踪和管理;
- 测试计划、目标、用例、过程和脚本;
- 分析、评审、测试结果和报告;
- 工作产品分析;

- 测试驱动器、模拟器、捕获/回放；
- 测试覆盖率分析；
- 缺陷和问题跟踪；
- 过程有效性分析和度量。

自动化支持的一个关键元素是用于所有测试交付物和工作产品的中心项目数据库。这可以指的是测试管理系统，包括用于保存、描述、文档化和跟踪测试，并且记录和跟踪评审及测试目标和结果的辅助设施。好的工具可以使得这些信息很容易被项目组获得，并且提供稳定的工作流支持来简化和跟踪软件开发过程。其他一些重要的工具支持包括：用于支持不同测试环境的测试床和模拟器；提供变更前后分析和工作产品风险及复杂度评价的静态分析器和比较器；用于测试执行和回归的测试驱动及捕获/回放工具；度量和报告测试结果及覆盖率的动态分析工具等。

测试自动化可以带来一些好处，包括：

- 测试的可重复性和一致性；
- 测试可扩展且可复用；
- 可以形成一个现实的测试基线套；
- 弥补测试环境的不足或不可用；
- 节省人力。

然而，这些好处并不是容易看到的。工具本身易于使用并不意味着就能成功。测试工具本身能做的东西很少，关键在于使用工具的人。很多刚拥有测试工具的人经常过分夸大工具的功效，并投入太高的期望。然而，测试工具并不是“银弹(Silver Bullet, 主要指一种可以解决任何问题的手段或工具)”。

成功的测试自动化有两个关键因素：

- 一个被很好理解的并且稳定的应用行为

你无法自动化那些没有很好定义的应用。那些还要进行大量需求修改和设计修改的应用使得测试自动化难以实现。即使一个相当稳定的应用，如果测试人员不了解它的行为或相关特定领域内的问题和需要，那么测试自动化也会充满问题。为了成功地测试自动化，你需要理解并预知应用的行为，然后才能按测试计划的方式用工具设计出测试。

- 一个专注的、有着丰富技能的测试项目组，并且被分配了足够的时间和资源

记住，测试件是软件。测试自动化应当像其他独立的项目支持管理一样被分配资源和时间。如果给你的测试项目组一个全新的测试工具，并且要求他们在空余的时间进行自动化，那么最终的结果将会是失败。

失败的测试自动化一般有如下表现：

- 买了一个测试工具，并且把该工具给了项目中的每个人；
- 假设自动化会节约项目时间和成本；
- 期望自动化会发现很多缺陷；
- 自动化没有理解的功能和特性；
- 自动化变化很大的功能和特性；
- 自动化不知道如何手工操作的测试；
- 没有把自动化作为一个项目来对待；

- 没有分配足够的资源给自动化;
- 缺乏自动化培训和支持;
- 忽略了测试自动化与开发过程的同步;
- 忽略了测试文档工作;
- 忽略了测试环境和配置;
- 忽略了测试的管理。

17.2.8 实践8：加强测试度量工作和缺陷分析工作，不断地改进测试

量化管理是项目管理的一个发展趋势。对于测试而言，加强测试成本、结果和效益的度量对测试的管理及改进是非常有帮助的。你无法管理你看不到和不理解的事情，并且对于理解测试来说，你必须收集和跟踪测试过程以及测试有效性方面的数据。

一般来说，在测试过程中，我们需要度量的基本数据包括：

- 测试投入的工作量和成本数据;
- 测试任务完成情况;
- 测试规模数据;
- 测试结果数据，包括缺陷数据，覆盖率数据等。

在基本数据度量的基础上，我们可以进行一些数据的分析，包括：

- 测试用例密度(相对代码行或功能点);
- 测试用例开发效率(测试用例总数/测试开发投入总工作量);
- 测试用例新增比例(新增用例数/测试用例总数);
- 测试用例自动化比例(自动化用例数/测试用例总数);
- 测试用例有效性(测试用例发现缺陷数/测试用例总数，或者测试用例发现缺陷数/缺陷总数);

注：在软件工程中有个 2/8 规律，使用在这里即 80% 的缺陷可以用 20% 的用例来发现。因此从项目的投入产出看，希望能够花最小的代价获得最大的好处。然而，对于质量要求很高的产品来说，这会抑制测试人员设计更广泛的用例。

- 测试用例执行效率(测试用例执行总数/测试执行投入总工作量);
- 测试漏测比率(版本发布后用户发现的缺陷数/版本发布前发现的缺陷总数);
- 每个版本的测试用例通过率(通过的测试用例数/计划执行的测试用例总数);
- 缺陷数据随版本变化的发展趋势;
- 缺陷数据随时间变化的发展趋势;
- 缺陷数据与测试投入工作量比较数据随时间变化情况;
- 缺陷数据与投入执行测试用例比较数据随时间变化情况;
- 缺陷数据与新增用例比较数据随时间变化情况;
- 模块或子系统缺陷分布情况分析;
- 测试计划完成比率。

有了充分的度量数据，管理人员就有了更好调整测试的依据，同时也为今后类似的项目提供了参考。但是有一点需要谨记：度量绝对不能用于对个人或组织的考核！^[6]

对于测试来说,进行漏测分析有助于测试的不断改进。漏测的分析不仅仅分析版本发布之后的缺陷,还可以针对内部每轮系统测试版本的漏测问题进行分析。一般对于每个缺陷我们需要从以下几个角度进行分析:

- 该缺陷是否能够在内部或者上一个系统测试版本中被发现?为什么没有被发现?如何避免这类情况产生?
- 该缺陷是否有相应的用例?如果没有,则设计用例同时还需要分析是否有类似的问题?针对这些类似的问题是否也需要补充相应的用例?
- 该缺陷是否属于因开发修改其他缺陷而引入的新缺陷?为什么会引入新的缺陷?回归时为什么没有考虑这方面的测试,是否回归分析不够全面?如何改进?

17.2.9 实践9:加强测试的培训并且为测试人员提供技能发展的通道

在软件测试中存在一个很大的误区,即很多管理人员把不合格的开发人员安排做测试,这导致了一种不利的思维:即测试是不重要的,测试是不需要技术的工作。因而很多人把测试作为一个向开发过渡的工作^[4]。然而,事实上,安排一个没有丰富开发经验的人来做测试工作是不利的。因为要做好测试工作,你必须对开发很了解。而另一方面,测试本身是一门需要技术的学问,它包含了众多的理论和实践。缺乏这些知识和经验,测试的深度和广度就不够,测试的质量也就无法保证。因此,对测试人员加强培训是很关键的。

测试和开发有很多的不同,在很多公司,对于开发人员的技术发展通道一般都比较明确,然而对测试人员的技术发展通道却比较模糊,这在很大程度上打击了测试人员的积极性和士气。很多测试人员看不到未来的发展方向,也定位不准自己的角色。这导致的结果是大量测试人员离职或转去做开发工作,而对公司来说,整体的测试水平一直得不到提高。

17.2.10 实践10:加强沟通和交流,让项目组内所有人员都了解测试的工作及其重要性

在很多时候,我们经常会听到这样的争吵。

开发人员:“这个不是问题,上次需求早就改过了,你怎么还提问题单?”

测试人员:“我怎么不知道?你们改需求怎么不通知测试?”

开发人员:“只是一些小的特性变化,没有开专门会议,大家都是通过邮件讨论的,可能没有抄送你们测试?”

测试人员:“这是不允许,没有测试人员的同意是不能算数的。”

项目经理:“好了,下不为例,下次讨论时一定要把测试叫上。”

然而,下次类似的事情还会继续上演。这只是开发和测试在沟通和交流上脱节的一个例子,你也许还可以举出很多例子。在开发过程中,测试是作为开发的一个下游部门,这是一个不争的事实,在很多公司中,尽管有相应的过程来要求开发和测试之间的信息互通,但是由于对测试的不够重视,经常出现测试人员最后知道项目变更的情况。这一方面造成了测试很多工作的浪费,另一方面也给测试无形中造成了更紧张的进度压力。这是一

个现实的情况，这种情况的彻底改变需要由上而下的变更。只有当管理层充分认识到测试的重要性，以及测试的工作内容时，测试才可能处在和开发同等的位置上，享有同样的权利，并获得充分的资源和时间。

这个变化依赖于管理者的真知灼见，在很多情况下只能通过测试人员自己去争取，这个过程是缓慢且不彻底的。然而却是没有办法中的一个好的方法。因此，很多有丰富测试经验的人员认为：作为一名测试人员，需要有着比开发人员更好的沟通能力。测试人员经常需要主动去和开发人员沟通，询问文档的完成情况，设计的变更情况等；经常需要和管理人员进行沟通，让管理人员了解测试的工作是有效的，不能越过的，并且还需要管理人员继续大力支持，包括资源的支持和时间的支持；经常需要说服管理人员，版本在没有经过完全测试之前发布出去是有风险的等。

加强测试与开发的沟通有助于打开测试与开发之间的隔阂，并有效地配合彼此的工作，提高产品的质量。

17.3 本章小结

本章列出了一些在测试工作中需要参考的原则和一些可供使用的最佳实践。这些经验一方面来自作者本身的测试经验，另一方面总结于业界一些测试专家的意见。这些原则和实践并不是覆盖全部的，你可能会发现一些其他更有效的原则和实践。如果有，请多与笔者进行交流，让我们一起来促进测试技术的发展。

附录 A 常见测试术语

表 A-1 测试术语表

英文名	中文名	简介
Acceptance testing	可接受性测试	一般由用户/客户进行的确认是否可以接受一个产品的验证性测试
Actual outcome	实际结果	被测对象在特定的条件下实际产生的结果
Ad Hoc testing	随机测试	测试人员通过随机的尝试系统的功能, 试图使系统中断
Algorithm	算法	(1) 一个定义好的有限规则集, 用于在有限步骤内解决一个问题; (2) 执行一个特定任务的任何操作序列
Algorithm analysis	算法分析	一个软件的验证确认任务, 用于保证选择的算法是正确的、合适的和稳定的, 并且满足所有精确性、规模和时间方面的要求
Alpha testing	Alpha 测试	由选定的用户进行的产品早期性测试。这个测试一般是在可控制的环境下进行的
analysis	分析	(1) 分解到一些原子部分或基本原则, 以便确定整体的特性; (2) 一个推理的过程, 显示一个特定的结果是假设前提的结果; (3) 一个问题的方法研究, 并且问题被分解为一些小的相关单元作进一步详细研究
Anomaly	异常	在文档或软件操作中观察到的任何与期望违背的结果
Application software	应用软件	满足特定需要的软件
Arc testing	弧形测试	参考分支测试(Branch Testing)
Architecture	构架	一个系统或组件的组织结构
ASQ	自动化软件质量	Automated Software Quality, 使用软件工具来提高软件的质量
Assertion	断言	指定一个程序必须已经存在的状态的一个逻辑表达式, 或者一组程序变量在程序执行期间的某个点上必须满足的条件
Assertion checking	断言检查	用户在程序中嵌入的断言的检查
Audit	审计	一个或一组工作产品的独立检查以评价与规格、标准、契约或其他准则的符合程度
Audit trail	审计跟踪	系统审计活动的一个时间记录
Automated testing	自动化测试	使用自动化测试工具来进行测试, 这类测试一般不需要人干预, 通常在 GUI、性能等测试中用得较多
Backus-Naur Form	BNF 范式	一种分析语言, 用于形式化描述语言的语法
Baseline	基线	一个已经被正式评审和批准的规格或产品, 它作为进一步开发的一个基础, 并且必须通过正式的变更流程来变更

续表

英文名	中文名	简介
Basic Block	基本块	一个或多个顺序的可执行语句块, 不包含任何分支语句
Basis test set	基本测试集	根据代码逻辑引出来的一个测试用例集合, 它保证能获得100%的分支覆盖
Behaviour	行为	对于一个系统的一个函数的输入和预置条件组合以及需要的反应, 一个函数的所有规格包含一个或多个行为
Benchmark	标杆/指标/基准	一个标准, 根据该标准可以进行度量或比较
Beta testing	Beta 测试	在客户场地, 由客户进行的对产品预发布版本的测试, 这个测试一般是不可控的
Big-bang testing	大锤测试/一次性集成测试	非渐增式集成测试的一种策略, 测试的时候把所有系统的组件一次性组合成系统进行测试
Black box testing	黑盒测试	根据软件的规格对软件进行的测试, 这类测试不考虑软件内部的运作原理, 因此软件对用户来说就像一个黑盒子
Bottom-up testing	由低向上测试	渐增式集成测试的一种, 其策略是先测试底层的组件, 然后逐步加入较高层次的组件进行测试, 直到系统所有组件都加入到系统
Boundary value	边界值	一个输入或输出值, 它处在等价类的边界上
Boundary value coverage	边界值覆盖	通过测试用例, 测试组件等价类的所有边界值
Boundary value testing	边界值测试	通过边界值分析方法来生成测试用例的一种测试策略
Boundry value analysis	边界值分析	该分析一般与等价类一起使用。经验认为软件的错误经常在输入的边界上产生, 因此边界值分析就是分析软件输入边界的一种方法
Branch	分支	在组件中, 控制从任何语句到其他任何非直接后续语句的一个条件转换, 或者是一个无条件转换
Branch condition	分支条件	参考判定条件 (decision condition)
Branch condition combination coverage	分支条件组合覆盖	在每个判定中所有分支条件结果组合被测试用例覆盖到的百分比
Branch condition combination testing	分支条件组合测试	通过执行分支条件结果组合来设计测试用例的一种方法
Branch condition coverage	分支条件覆盖	每个判定中分支条件结果被测试用例覆盖到的百分比
Branch condition testing	分支条件测试	通过执行分支条件结果来设计测试用例的一种方法
Branch coverage	分支覆盖	通过测试执行到的分支的百分比
Branch outcome	分支结果	见判定结果 (decision outcome)
Branch point	分支点	见判定 (decision)
Branch testing	分支测试	通过执行分支结果来设计测试用例的一种方法

续表

英文名	中文名	简介
Breadth testing	广度测试	在测试中测试一个产品的所有功能, 但是不测试更细节的特性
Bug	缺陷	参考 Fault
Capture/playback tool	捕获/回放工具	参考 capture/replay tool
Capture/Replay tool	捕获/回放工具	一种测试工具, 能够捕获在测试过程中传递给软件的输入, 并且能够在以后的时间中, 重复这个执行的过程。这类工具一般在 GUI 测试中用得较多
CASE	计算机辅助软件工程	computer aided software engineering, 用于支持软件开发的一个自动化系统
CAST	计算机辅助测试	在测试过程中使用计算机软件工具进行辅助的测试
Cause-effect graph	因果图	一个图形, 用来表示输入(原因)与结果之间的关系, 可以被用来设计测试用例
Certification	证明	一个过程, 用于确定一个系统或组件与特定的需求相一致
Change control	变更控制	一个用于计算机系统或系统数据修改的过程, 该过程是质量保证程序的一个关键子集, 需要被明确地描述
Code audit	代码审计	由一个人、组或工具对源代码进行的一个独立的评审, 以验证其与设计规格、程序标准的一致性。正确性和有效性也会被评价
Code coverage	代码覆盖率	一种分析方法, 用于确定在一个测试套执行后, 软件的哪些部分被执行到了, 哪些部分没有被执行到
Code inspection	代码检视	一个正式的同行评审手段, 在该评审中, 作者的同行根据检查表对程序的逻辑进行提问, 并检查其与编码规范的一致性
Code walkthrough	代码走读	一个非正式的同行评审手段, 在该评审中, 代码被使用一些简单的测试用例进行人工执行, 程序变量的状态被手工分析, 以分析程序的逻辑和假设
Code-based testing	基于代码的测试	根据从实现中引出的目标设计测试用例
Coding standards	编程规范	一些编程方面需要遵循的标准, 包括命名方式、排版格式等内容
Compatibility testing	兼容性测试	测试软件是否和系统的其他与之交互的元素之间兼容, 如: 浏览器、操作系统、硬件等
Complete path testing	完全路径测试	参考穷尽测试(exhaustive testing)
Completeness	完整性	实体的所有必需部分必须被包含的属性
Complexity	复杂性	系统或组件难于理解或验证的程度
Component	组件	一个最小的软件单元, 有着独立的规格
Component testing	组件测试	参考单元测试

续表

英文名	中文名	简介
Computation data use	计算数据使用	一个不在条件中的数据使用
Computer system security	计算机系统安全性	计算机软件和硬件对偶然的或故意的访问、使用、修改或破坏的一种保护机制
Condition	条件	一个不包含布尔操作的布尔表达式, 例如, $A < B$ 是条件, 但 $A \text{ and } B$ 不是条件
Condition coverage	条件覆盖	通过测试执行到的条件百分比
Condition outcome	条件结果	条件为真为假的评价
Configuration control	配置控制	配置管理的一个方面, 包括评价、协调、批准和实现配置项的变更
Configuration management	配置管理	一套技术和管理方面的原则用于确定和文档化一个配置项的功能和物理属性、控制对这些属性的变更、记录和报告变更处理和实现的状态以及验证与指定需求的一致性
Conformance criterion	一致性标准	判断组件在一个特定输入值上的行为是否符合规格的一种方法
Conformance Testing	一致性测试	测试一个系统的实现是否和其基于的规格相一致的测试
Consistency	一致性	在系统或组件的各组成部分和文档之间没有矛盾, 一致的程度
Consistency checker	一致性检查器	一个软件工具, 用于测试设计规格中需求的一致性和完整性
Control flow	控制流	程序执行中所有可能的事件顺序的一个抽象表示
Control flow graph	控制流图	通过一个组件的可能替换控制流路径的一个图形表示
Conversion testing	转换测试	用于测试已有系统的数据是否能够转换到替代系统上的一种测试
Corrective maintenance	故障检修	用于纠正硬件或软件中故障的维护
Correctness	正确性	软件在其规格、设计和编码中没有故障的程度。软件、文档和其他项满足需求的程度。软件、文档和其他项满足用户明显的和隐含的需求的程度
Coverage	覆盖率	用于确定测试所执行到的覆盖项的百分比
Coverage item	覆盖项	作为测试基础的一个入口或属性, 如语句、分支、条件等
Crash	崩溃	计算机系统或组件突然并完全地丧失功能
Criticality	关键性	需求、模块、错误、故障、失效或其他项对一个系统的操作或开发影响的程度
Criticality analysis	关键性分析	需求的一种分析, 它根据需求的风险情况给每个需求项分配一个关键级别
C-use	C-use	参考计算数据使用(computation data use)

续表

英文名	中文名	简介
Cyclomatic complexity	循环复杂度	一个程序中独立路径的数量
Data corruption	数据污染	违背数据一致性的情况
Data definition	数据定义	一个可执行语句, 在该语句上, 一个变量被赋予了一个值
Data definition C-use coverage	数据定义 C-use 覆盖	在组件中被测试执行到的数据定义 C-use 使用对的百分比
Data definition C-use pair	数据定义 C-use 使用对	一个数据定义和一个计算数据使用, 数据使用的值是数据定义的值
Data definition P-use coverage	数据定义 P-use 覆盖	在组件中被测试执行到的数据定义 P-use 使用对的百分比
Data definition P-use pair	数据定义 P-use 使用对	一个数据定义和一个条件数据使用, 数据使用的值是数据定义的值
Data definition-use coverage	数据定义使用覆盖	在组件中被测试执行到的数据定义使用对的百分比
Data definition-use pair	数据定义使用对	一个数据定义和一个数据使用, 数据使用的值是数据定义的值
Data definition-use testing	数据定义使用测试	以执行数据定义使用对为目标进行测试用例设计的一种技术
Data dictionary	数据字典	(1) 一个软件系统中使用的所有数据项名称, 以及这些项相关属性的集合。(2) 数据流、数据元素、文件、数据基础和相关处理的一个集合
Data flow analysis	数据流分析	一个软件验证和确认过程, 用于保证输入和输出数据以及它们的格式是被适当定义的, 并且数据流是正确的
Data flow coverage	数据流覆盖	测试覆盖率的度量是根据变量在代码中的使用情况
Data flow diagram	数据流图	把数据源、数据接受、数据存储和数据处理作为节点描述的一个图形, 数据之间的逻辑体现为节点之间的边
Data flow testing	数据流测试	根据代码中变量的使用情况进行的测试
Data integrity	数据完整性	一个数据集合完全、正确和一致的程度
Data use	数据使用	一个可执行的语句, 在该语句中, 变量的值被访问
Data validation	数据确认	用于确认数据不正确、不完整和不合理的过程
Dead code	死代码	在程序操作过程中永远不可能被执行到的代码
Debugging	调试	发现和去除软件失效根源的过程
Decision	判定	一个程序控制点, 在该控制点上, 控制流有两个或多个可替换路由
Decision condition	判定条件	判定内的一个条件
Decision coverage	判定覆盖	在组件中被测试执行到的判定结果的百分比
Decision outcome	判定结果	一个判定的结果, 决定控制流走哪条路径

续表

英文名	中文名	简介
Decision table	判定表	一个表格, 用于显示条件和条件导致动作的集合
Depth testing	深度测试	执行一个产品的一个特性的所有细节, 但不测试所有特性, 比较广度测试
Design of experiments	实验设计	一种计划实验的方法, 这样适合分析的数据可以被收集
Design-based testing	基于设计的测试	根据软件的构架或详细设计引出测试用例的一种方法
Desk checking	桌面检查	通过手工模拟软件执行的方式进行测试的一种方式
Diagnostic	诊断	检测和隔离故障或失效的过程
Dirty testing	肮脏测试	参考负面测试(negative testing)
Disaster recovery	灾难恢复	一个灾难的恢复和重建过程或能力
Documentation testing	文档测试	测试关注于文档的正确性
Domain	域	值被选择的一个集合
Domain testing	域测试	参考等价划分测试(equivalence partition testing)
Dynamic analysis	动态分析	根据执行的行为评价一个系统或组件的过程
Dynamic Testing	动态测试	通过执行软件的手段来测试软件
embedded software	嵌入式软件	软件运行在特定硬件设备中, 不直接与用户交互。这类系统一般要求实时性较高
Emulator	仿真	一个模仿另一个系统的系统或设备, 它接受相同的输入并产生相同的输出
End-to-End testing	端到端测试	在一个模拟现实使用的场景下测试一个完整的应用环境, 例如和数据库交互, 使用网络通信等
Entity relationship diagram	实体关系图	描述现实世界中实体及它们关系的图形
Entry point	入口点	一个组件的第一个可执行语句
Equivalence Class	等价类	组件输入或输出域的一个部分, 在该部分中, 组件的行为从组件的规格上来看认为是相同的
Equivalence partition coverage	等价划分覆盖	在组件中被测试执行到的等价类的百分比
Equivalence partition testing	等价划分测试	根据等价类设计测试用例的一种技术
Equivalence Partitioning	等价划分	组件的一个测试用例设计技术, 该技术从组件的等价类中选取典型的点进行测试
Error	错误	IEEE 的定义是: 一个人为产生不正确结果的行为
Error guessing	错误猜测	根据测试人员以往的经验猜测可能出现问题的地方来进行用例设计的一种技术

续表

英文名	中文名	简介
Error seeding	错误播种/错误插值	故意插入一些已知故障(fault)到一个系统中去的过程,目的是为了根据错误检测和跟踪的效率并估计系统中遗留缺陷的数量
Exception	异常/例外	一个引起正常程序执行挂起的事件
Executable statement	可执行语句	一个语句在被编译后会转换成目标代码,当程序运行时会被执行,并且可能对程序数据产生动作
Exhaustive testing	穷尽测试	测试覆盖软件的所有输入和条件组合
Exit point	出口点	一个组件的最后一个可执行语句
Expected outcome	期望结果	参考预期结果(predicted outcome)
Failure	失效	软件的行为与其期望的服务相背离
Fault	故障	在软件中一个错误的表现
Feasible path	可达路径	可以通过一组输入值和条件执行到的一条路径
Feature testing	特性测试	参考功能测试(Functional Testing)
FMEA	失效模型效果分析	Failure Modes and Effects Analysis, 可靠性分析中的一种方法,用于在基本组件级别上确认对系统性能有重大影响的失效
FMECA	失效模型效果关键性分析	Failure Modes and Effects Criticality Analysis, FMEA 的一个扩展,它分析了失效结果的严重性
FTA	故障树分析	Fault Tree Analysis, 引起一个不需要事件产生的条件和因素的确认和分析,通常是严重影响系统性能、经济性、安全性或其他需要特性
Functional decomposition	功能分解	参考模块分解(modular decomposition)
Functional specification	功能规格说明书	一个详细描述产品特性的文档
Functional testing	功能测试	测试一个产品的特性和可操作行为以确定它们满足规格
Glass box testing	玻璃盒测试	参考白盒测试(White Box Testing)
IEEE	IEEE	Institute of Electrical and Electronic Engineers
Incremental testing	渐增测试	集成测试的一种,组件逐渐被增加到系统中直到整个系统被集成
Infeasible path	不可达路径	不能够通过任何可能的输入值集合执行到的路径
Input domain	输入域	所有可能输入的集合
Inspection	检视	对文档进行的一种评审形式
Installability testing	可安装性测试	确定系统的安装程序是否正确的测试
Instrumentation	插装	在程序中插入额外的代码以获得程序在执行时行为的信息
Instrumenter	插装器	执行插装的工具

续表

英文名	中文名	简介
Integration testing	集成测试	测试一个应用组合后的部分以确保它们的功能在组合之后正确。该测试一般在单元测试之后进行,用于揭露集成组件之间接口和交互故障的一种测试
Interface	接口	两个功能单元的共享边界
Interface analysis	接口分析	分析软件与硬件、用户和其他软件之间接口的需求规格
Interface testing	接口测试	测试系统组件间接口的一种测试
Invalid inputs	无效输入	在程序功能输入域之外的测试数据
Isolation testing	孤立测试	组件测试(单元测试)策略中的一种,把被测组件从其上下文组件之中孤立出来,通过设计驱动和桩进行测试的一种方法
Job	工作	一个用户定义要计算机完成的工作单元
Job control language	工作控制语言	用于确定工作顺序,描述它们对操作系统要求并控制它们执行的语言
LCSAJ	线性代码顺序和跳转	Linear Code Sequence And Jump, 包含三个部分:可执行语句线性顺序的起始,线性顺序的结束,在线性顺序结束处控制流跳转的目标语句
LCSAJ coverage	LCSAJ 覆盖	在组件中被测试执行到的 LCSAJ 的百分比
LCSAJ testing	LCSAJ 测试	根据 LCSAJ 设计测试用例的一种技术
Load Testing	负载测试	通过测试系统在资源超负荷情况下的表现,以发现设计上的错误或验证系统的负载能力
Logic analysis	逻辑分析	(1)评价软件设计的关键安全方程式、算法和控制逻辑的方法。(2)评价程序操作的顺序并且检测可能导致灾难的错误
Logic-coverage testing	逻辑覆盖测试	参考结构化测试用例设计(structural test case design)
Maintainability	可维护性	一个软件系统或组件可以被修改的容易程度,这个修改一般是因为缺陷纠正、性能改进或特性增加引起的
Maintainability testing	可维护性测试	测试系统是否满足可维护性目标
Modified condition/decision coverage	修改条件/判定覆盖	在组件中被测试执行到的修改条件/判定的百分比
Modified condition/decision testing	修改条件/判定测试	根据 MC/DC 设计测试用例的一种技术
Monkey testing	跳跃式测试	随机性,跳跃式的测试一个系统,以确定一个系统是否会崩溃
MTBF	平均失效间隔 实际	mean time between failures, 两次失效之间的平均操作时间
MTTF	平均失效时间	mean time to failure, 第一次失效之前的平均时间
MTTR	平均修复时间	mean time to repair, 两次修复之间的平均时间

续表

英文名	中文名	简介
Multiple condition coverage	多条件覆盖	参考分支条件组合覆盖(branch condition combination coverage)
Mutation analysis	变体分析	一种确定测试用例套完整性的方法,该方法通过判断测试用例套能够区别程序与其变体之间的程度
Negative Testing	逆向测试/反向测试/负面测试	测试瞄准于使系统不能工作
Non-functional requirements testing	非功能性需求测试	与功能不相关的需求测试,如:性能测试、可用性测试等
N-switch coverage	N 切换覆盖	在组件中被测试执行到的 N 转换顺序的百分比
N-switch testing	N 切换测试	根据 N 转换顺序设计测试用例的一种技术,经常用于状态转换测试中
N-transitions	N 转换	N + 1 转换顺序
Operational testing	可操作性测试	在系统或组件操作的环境中评价它们的表现
Output domain	输出域	所有可能输出的集合
Partition testing	分类测试	参考等价划分测试(equivalence partition testing)
Path	路径	一个组件从入口到出口的一条可执行语句顺序
Path coverage	路径覆盖	在组件中被测试执行到的路径的百分比
Path sensitizing	路径敏感性	选择一组输入值强制组件走一个给定的路径
Path testing	路径测试	根据路径设计测试用例的一种技术,经常用于状态转换测试中
Performance testing	性能测试	测试软件在集成系统中的运行性能,该测试经常与负载测试一起进行
Performance testing	性能测试	评价一个产品或组件与性能需求是否符合的测试
Portability testing	可移植性	测试瞄准于证明软件可以被移植到指定的硬件或软件平台上
Positive Testing	正向测试	测试瞄准于显示系统能够正常工作
Precondition	预置条件	环境或状态条件,组件执行之前必须被填充一个特定的输入值
Predicate	谓词	一个逻辑表达式,结果为“真”或“假”
Predicate data use	谓词数据使用	在谓词中的一个数据使用
Program instrumenter	程序插装	参考插装(instrumenter)
Progressive testing	递进测试	在先前特性回归测试之后对新特性进行测试的一种策略
Pseudo-random	伪随机	看似随机的,实际上是根据预先安排的顺序进行的
P-use	P-use	参考条件数据使用(predicate data use)

续表

英文名	中文名	简介
QA	质量保证	quality assurance, (1)已计划的系统性活动, 用于保证一个组件、模块或系统遵从已确立的需求。(2)采取所有的活动以保证一个开发组织交付的产品满足性能需求和已确立的标准和过程
QC	质量控制	quality control, 用于获得质量需求的操作技术和过程, 如测试活动
Race Condition	竞争状态	并行问题的根源。对一个共享资源的多个访问, 至少包含了一个写操作, 但是没有有一个机制来协调同时发生的访问
Recovery testing	恢复性测试	验证系统从失效中恢复能力的测试
Regression analysis and testing	回归分析和测试	一个软件验证和确认任务以确定在修改后需要重复测试和分析的范围
Regression testing	回归测试	在发生修改之后重新测试先前的测试以保证修改的正确性
Release	发布	一个批准版本的正式通知和分发
Reliability	可靠性	一个系统或组件在规定的条件下在指定的时间内执行其需要功能的能力
Reliability assessment	可靠性评价	确定一个已有系统或组件的可靠性级别的过程
Requirements-based testing	基于需求的测试	根据软件组件的需求导出测试用例的一种设计方法
Review	评审	在产品开发过程中, 把产品提交给项目成员、用户、管理者或其他相关人员评价或批准的过程
Risk	风险	不期望效果的可能性和严重性的一个度量
Risk assessment	风险评估	对风险和风险影响的一个完整的评价
Safety	(生命)安全性	不会引起人员伤亡、产生疾病、毁坏或损失设备和财产、或者破坏环境
Safety critical	严格的安全性	一个条件、事件、操作、过程或项, 它的认识、控制或执行对生命安全的系统来说是非常关键的
Sanity Testing	理智测试	软件主要功能成分的简单测试以保证它是否能进行基本的测试。参考冒烟测试
SDP	软件开发计划	software development plan, 用于一个软件产品开发的项目计划
Security testing	安全性测试	验证系统是否符合安全性目标的一种测试
Security	(信息)安全性	参考计算机系统安全性(computer system security)
Serviceability testing	可服务性测试	参考可维护性测试(maintainability testing)
Simple subpath	简单子路径	控制流的一个子路径, 其中没有不必要的部分被执行
Simulation	模拟	使用另一个系统来表示一个物理的或抽象的系统的选定行为特性

续表

英文名	中文名	简介
Simulation	模拟	使用一个可执行模型来表示一个对象的行为
Simulator	模拟器	软件验证期间的一个设备、软件程序或系统, 当它给定一个控制的输入时, 表现的与一个给定的系统类似
SLA	服务级别协议	service level agreement, 服务提供商与客户之间的一个协议, 用于规定服务提供商应当提供什么服务
Smoke testing	冒烟测试	对软件主要功能进行快餐式测试。最早来自于硬件测试实践, 以确定新的硬件在第一次使用时不会着火
Software development process	软件开发过程	一个把用户需求转换为软件产品的开发过程
Software diversity	软件多样性	一种软件开发技术, 其中, 由不同的程序员或开发组开发的相同规格的不同程序, 目的是为了检测错误、增加可靠性
Software element	软件元素	软件开发或维护期间产生或获得的一个可交付的或过程内的文档
Software engineering	软件工程	一个应用于软件开发、操作和维护的系统性的、有纪律的、可量化的方法
Software engineering environment	软件工程环境	执行一个软件工程工作的硬件、软件和固件
Software life cycle	软件生命周期	开始于一个软件产品的构思, 结束于该产品不再被使用的这段期间
SOP	标准操作过程	standard operating procedures, 书面的步骤, 这对保证生产和处理的控制是必需的
Source code	源代码	用一种适合于输入到汇编器、编译器或其他转换设备的计算机指令和数据定义
Source statement	源语句	参考语句(statement)
Specification	规格	组件功能的一个描述, 格式是: 对指定的输入在指定的条件下的输出
Specified input	指定的输入	一个输入, 根据规格能预知其输出
Spiral model	螺旋模型	软件开发过程的一个模型, 其中的组成活动, 典型的包括需求分析、概要设计、详细设计、编码、集成和测试等活动被迭代的执行直到软件被完成
SQL	结构化查询语句	structured query language, 在一个关系数据库中查询和处理数据的一种语言
State	状态	一个系统、组件或模拟可能存在其中的一个条件或模式
State diagram	状态图	一个图形, 描绘一个系统或组件可能假设的状态, 并且显示引起或导致一个状态切换到另一个状态的事件或环境
State transition	状态转换	一个系统或组件的两个允许状态之间的切换
State transition testing	状态转换测试	根据状态转换来设计测试用例的一种方法
Statement	语句	程序语言的一个实体, 是典型的最小可执行单元

续表

英文名	中文名	简介
Statement coverage	语句覆盖	在一个组件中, 通过执行一定的测试用例所能达到的语句覆盖百分比
statement testing	语句测试	根据语句覆盖来设计测试用例的一种方法
Static Analysis	静态分析	分析一个程序的执行, 但是并不实际执行这个程序
Static Analyzer	静态分析器	进行静态分析的工具
Static Testing	静态测试	不通过执行来测试一个系统
Statistical testing	统计测试	通过使用对输入统计分布进行分析来构造测试用例的一种测试设计方法
Stepwise refinement	逐步优化	一个结构化软件设计技术, 数据和处理步骤首先被广泛地定义, 然后逐步地被进行了细化
Storage testing	存储测试	验证系统是否满足指定存储目标的测试
Stress Testing	压力测试	在规定的规格条件或者超过规定的规格条件下, 测试一个系统, 以评价其行为。类似负载测试, 通常是性能测试的一部分
Structural coverage	结构化覆盖	根据组件内部的结构度量覆盖率
Structural test case design	结构化测试用例设计	根据组件内部结构的分析来设计测试用例的一种方法
Structural testing	结构化测试	参考结构化测试用例设计(structural test case design)
Structured basis testing	结构化的基础测试	根据代码逻辑设计测试用例来获得 100% 分支覆盖的一种测试用例设计技术
Structured design	结构化设计	软件设计的任何遵循一定纪律的方法, 它按照特定的规则, 例如: 模块化, 由顶向下设计, 数据逐步优化, 系统结构和处理步骤
Structured programming	结构化编程	在结构化程序开发中的任何包含结构化设计和结果的软件开发技术
Structured walkthrough	结构化走读	参考走读(walkthrough)
Stub	桩	一个软件模块的框架或特殊目标实现, 主要用于开发和测试一个组件, 该组件调用或依赖这个模块
Symbolic evaluation	符号评价	参考符号执行(symbolic execution)
symbolic execution	符号执行	一个静态分析技术, 其中, 程序的执行用符号来模拟, 例如, 使用变量名而不是实际值, 程序的输出被表示成包含这些符号的逻辑或数学表达式
Symbolic trace	符号轨迹	一个计算机程序通过符号执行时经过的语句分支结果的一个记录
Syntax Testing	语法分析	根据输入语法来验证一个系统或组件的测试用例设计技术
System Analysis	系统分析	对一个计划的或现实的系统进行的一个系统性调查以确定系统的功能以及系统与其他系统之间的交互

续表

英文名	中文名	简介
System Design	系统设计	一个定义硬件和软件构架、组件、模块、接口和数据的过程以满足指定的规格
System Integration	系统集成	一个系统组件的渐增的连接和测试,直到一个完整的系统
System Testing	系统测试	从一个系统的整体而不是个体上来测试一个系统,并且该测试关注的是规格,而不是系统内部的逻辑
Technical Requirements Testing	技术需求测试	参考非功能需求测试(non-functional requirements testing)
Test Automation	测试自动化	使用工具来控制测试的执行、结果的比较、测试预置条件的设置、和其他测试控制和报告功能
Test Case	测试用例	用于特定目标而开发的一组输入、预置条件和预期结果
Test Case Design Technique	测试用例设计技术	选择和导出测试用例的技术
Test Case Suite	测试用例套	对被测软件的一个或多个测试用例的集合
Test Comparator	测试比较器	一个测试工具用于比较软件实际测试产生的结果与测试用例预期的结果
Test Completion Criterion	测试完成标准	一个标准用于确定被计划的测试何时完成
Test Coverage	测试覆盖	参考覆盖率(Coverage)
Test Driver	测试驱动	一个程序或测试工具用于根据测试套执行软件
Test Environment	测试环境	测试运行其上的软件和硬件环境的描述,以及任何其他与被测软件交互的软件,包括驱动和桩
Test Execution	测试执行	一个测试用例被被测软件执行,并得到一个结果
Test Execution Technique	测试执行技术	执行测试用例的技术,包括手工、自动化等
Test Generator	测试生成器	根据特定的测试用例产生测试用例的工具
Test Harness	测试用具	包含测试驱动和测试比较器的测试工具
Test Log	测试日志	一个关于测试执行所有相关细节的时间记录
Test Measurement Technique	测试度量技术	度量测试覆盖率的技术
Test Plan	测试计划	一个文档,描述了要进行的测试活动的范围、方法、资源和进度。它确定测试项、被测特性、测试任务、谁执行任务,并且任何风险都会与计划有冲突
Test Procedure	测试规程	一个文档,提供详细的测试用例执行指令
Test Records	测试记录	对每个测试,明确地记录被测组件的标识、版本、测试规格和实际结果
Test report	测试报告	一个描述系统或组件执行的测试和结果的文档
Test Script	测试脚本	一般指的是一个特定测试的一系列指令,这些指令可以被自动化测试工具执行

续表

英文名	中文名	简介
Test Specification	测试规格	一个文档, 用于指定一个软件特性、特性组合或所有特性的测试方法、输入、预期结果和执行条件
Test Strategy	测试策略	一个简单的高层文档, 用于描述测试的大致方法、目标和方向
Test suite	测试套	测试用例和/或测试脚本的一个集合, 与一个应用的特定功能或特性相关
Test target	测试目标	一组测试完成标准
Testability	可测试性	一个系统或组件有利于测试标准建立和确定这些标准是否被满足的测试执行的程度
Testing	测试	IEEE 给出的定义是: (1) 一个执行软件的过程, 以验证其满足指定的需求并检测错误。(2) 一个软件项的分析过程以检测已有条件之间的不同, 并评价软件项的特性
Thread testing	线程测试	自顶向下测试的一个变化版本, 其中, 递增的组件集成遵循需求子集的实现
Time sharing	时间共享	一种操作方式, 允许两个或多个用户在相同的计算机系统上同时执行计算机程序。其实现可能通过时间片轮转、优先级中断等
Top-down design	自顶向下设计	一种设计策略, 首先设计最高层的抽象和处理, 然后逐步向更低级别进行设计
Top-down testing	自顶向下测试	集成测试的一种策略, 首先测试最顶层的组件, 其他组件使用桩, 然后逐步加入较低层的组件进行测试, 直到所有组件被集成到系统中
Traceability	可跟踪性	开发过程的两个或多个产品之间关系可以被建立起来的程度, 尤其是产品彼此之间有一个前后处理关系
Traceability analysis	跟踪性分析	(1) 跟踪概念文档中的软件需求到系统需求; (2) 跟踪软件设计描述到软件需求规格, 以及软件需求规格到软件设计描述; (3) 跟踪源代码对应到设计规格, 以及设计规格对应到源代码。分析确定它们之间正确性、一致性、完整性、精确性的关系
Traceability matrix	跟踪矩阵	一个用于记录两个或多个产品之间关系的矩阵。例如, 需求跟踪矩阵是跟踪从需求到设计再到编码的实现
Transaction	事务/处理	(1) 一个命令、消息或输入记录, 它明确或隐含地调用了一个处理活动, 例如更新一个文件。(2) 用户和系统之间的一次交互。(3) 在一个数据库管理系统中, 完成一个特定目的的处理单元, 如恢复、更新、修改或删除一个或多个数据元素
Transform analysis	事务分析	系统的结构是根据分析系统需要处理的事务获得的一种分析技术
Trojan horse	特洛伊木马	一种攻击计算机系统的方法, 典型的方法是提供一个包含具有攻击性隐含代码的有用程序给用户, 在用户执行该程序的时候, 其隐含的代码对系统进行非法访问, 并可能产生破坏

续表

英文名	中文名	简介
Truth table	真值表	用于逻辑操作的一个操作表格
Unit Testing	单元测试	测试单个的软件组件,属于白盒测试范畴,其测试基础是软件内部的逻辑
Usability Testing	可用性测试	测试用户使用和学习产品的容易程度
Validation	确认	根据用户需要确认软件开发的产品的正确性
Verification	验证	评价一个组件或系统以确认给定开发阶段的产品是否满足该阶段开始时设定的标准
Version	版本	一个软件项或软件元素的一个初始发布或一个完整的再发布
Volume testing	容量测试	使用大容量数据测试系统的一种策略
Walkthrough	走读	一个针对需求、设计或代码的非正式的同行评审,一般由作者发起,由作者的同行参与进行的评审过程
Waterfall model	瀑布模型	软件开发过程模型的一种,包括概念阶段、需求阶段、设计阶段、实现阶段、测试阶段、安装和检查阶段、操作和维护阶段,这些阶段按次序进行,可能有部分重叠,但很少会迭代
White Box Testing	白盒测试	根据软件内部的工作原理分析来进行测试

表 A-2 中英文对照表

中文名	英文名	中文名	英文名
Alpha 测试	Alpha Testing	不可达路径	infeasible path
Beta 测试	Beta Testing	测试	Testing
BNF 范式	Backus- Naur Form	测试报告	Test report
C-use	C-use	测试比较器	Test comparator
IEEE	IEEE	测试策略	Test strategy
LCSAJ 测试	LCSAJ testing	测试度量技术	Test measurement technique
LCSAJ 覆盖	LCSAJ coverage	测试覆盖	Test coverage
N 切换测试	N-switch testing	测试规程	Test procedure
N 切换覆盖	N-switch coverage	测试规格	Test Specification
N 转换	N-transitions	测试环境	Test environment
P-use	P-use	测试计划	Test Plan
安全性(生命)	Safety	测试记录	Test records
安全性(信息)	security	测试脚本	test Script
安全性测试	security testing	测试目标	test target
肮脏测试	dirty testing	测试驱动	test driver

续表

中文名	英文名	中文名	英文名
白盒测试	White Box Testing	测试日志	test log
版本	Version	测试生成器	test generator
崩溃	Crash	测试套	test suite
边界值	boundary value	测试完成标准	test completion criterion
边界值测试	boundary value testing	测试用具	test harness
边界值分析	Boundry Value Analysis	测试用例	test case
边界值覆盖	boundary value coverage	测试用例设计技术	test case design technique
编程规范	coding standards	测试用例套	test case suite
变更控制	change control	测试执行	test execution
变体分析	mutation analysis	测试执行技术	test execution technique
标杆/指标/基准	Benchmark	测试自动化	test automation
标准操作过程	SOP	插装	instrumentation
玻璃盒测试	glass box testing	插装器	instrumenter
捕获/回放工具	capture/playback tool	程序插装	program instrumenter
捕获/回放工具	Capture/Replay Tool	出口点	exit point
存储测试	storage testing	分支条件组合覆盖	branch condition combination coverage
错误	error	风险	risk
错误播种/错误插值	error seeding	风险评估	risk assessment
错误猜测	error guessing	服务级别协议	SLA
大锤测试/一次性集成测试	big-bang testing	符号轨迹	symbolic trace
代码覆盖率	Code Coverage	符号评价	symbolic evaluation
代码检视	Code Inspection	符号执行	symbolic execution
代码审计	code audit	符号执行	symbolic execution
代码走读	Code Walkthrough	负载测试	Load Testing
单元测试	Unit Testing	复杂性	complexity
等价划分	Equivalence Partitioning	覆盖率	coverage
等价划分测试	equivalence partition testing	覆盖项	coverage item
等价划分覆盖	equivalence partition coverage	跟踪矩阵	traceability matrix
等价类	Equivalence Class	跟踪性分析	traceability analysis
递进测试	progressive testing	工作	Job

续表

中文名	英文名	中文名	英文名
调试	Debugging	工作控制语言	job control language
动态测试	Dynamic Testing	功能测试	Functional Testing
动态分析	dynamic analysis	功能分解	functional decomposition
端到端测试	End-to-End testing	功能规格说明书	Functional Specification
断言	assertion	构架	architecture
断言检查	assertion checking	孤立测试	isolation testing
多条件覆盖	multiple condition coverage	故障	fault
发布	release	故障检修	corrective maintenance
仿真	emulator	故障树分析	FTA
非功能性需求测试	non-functional requirements testing	关键性	criticality
分类测试	partition testing	关键性分析	criticality analysis
分析	analysis	广度测试	Breadth Testing
分支	branch	规格	specification
分支测试	branch testing	黑盒测试	Black Box Testing
分支点	branch point	弧形测试	arc testing
分支覆盖	branch coverage	恢复性测试	recovery testing
分支结果	branch outcome	回归测试	Regression Testing
分支条件	branch condition	回归分析和测试	regression analysis and testing
分支条件测试	branch condition testing	基本测试集	basis test set
分支条件覆盖	branch condition coverage	基本块	Basic Block
分支条件组合测试	branch condition combination testing	基线	baseline
基于代码的测试	code-based testing	可移植性	portability testing
基于设计的测试	design-based testing	可用性测试	Usability Testing
基于需求的测试	requirements-based testing	可执行语句	executable statement
集成测试	integration testing	控制流	control flow
控制流图	control flow graph	计算机辅助测试	CAST
理智测试	Sanity Testing	计算机辅助软件工程	CASE
路径	path	计算机系统安全性	computer system security
路径测试	path testing	计算数据使用	computation data use
路径覆盖	path coverage	技术需求测试	technical requirements testing

续表

中文名	英文名	中文名	英文名
路径敏感性	path sensitizing	兼容性测试	Compatibility Testing
逻辑分析	logic analysis	检视	inspection
逻辑覆盖测试	logic-coverage testing	简单子路径	simple subpath
螺旋模型	spiral model	渐增测试	incremental testing
冒烟测试	Smoke Testing	接口	interface
模拟	simulation	接口测试	interface testing
接口分析	interface analysis	模拟器	simulator
结构化编程	structured programming	逆向测试/反向测试/负面测试	Negative Testing
结构化测试	structural testing	判定	decision
结构化测试用例设计	structural test case design	判定表	Decision table
结构化查询语句	SQL	判定覆盖	Decision coverage
结构化的基础测试	structured basis testing	判定结果	Decision outcome
结构化覆盖	structural coverage	判定条件	Decision condition
结构化设计	structured design	配置管理	configuration management
结构化走读	structured walkthrough	配置控制	configuration control
竞争状态	Race Condition	平均失效间隔实际	MTBF
静态测试	Static Testing	平均失效时间	MTTF
静态分析	Static Analysis	平均修复时间	MTTR
静态分析器	Static Analyzer	评审	review
可安装性测试	installability testing	瀑布模型	waterfall model
可操作性测试	operational testing	期望结果	expected outcome
可测试性	testability	嵌入式软件	embedded software
可达路径	feasible path	穷尽测试	Exhaustive Testing
可服务性测试	serviceability testing	缺陷	bug
可跟踪性	traceability	确认	validation
可接受性测试	Acceptance Testing	容量测试	volume testing
可靠性	reliability	入口点	entry point
可靠性评价	reliability assessment	软件多样性	software diversity
可维护性	maintainability	软件工程	software engineering
可维护性测试	maintainability testing	软件工程环境	software engineering environment

续表

中文名	英文名	中文名	英文名
软件开发过程	software development process	算法	algorithm
软件开发计划	SDP	算法分析	algorithm analysis
软件生命周期	software life cycle	随机测试	Ad Hoc Testing
软件元素	software element	特洛伊木马	trojan horse
深度测试	Depth Testing	特性测试	feature testing
审计	audit	条件	condition
审计跟踪	audit trail	条件覆盖	condition coverage
失效	failure	条件结果	condition outcome
失效模型效果分析	FMEA	跳跃式测试	Monkey Testing
失效模型效果关键性分析	FMECA	统计测试	statistical testing
时间共享	time sharing	完全路径测试	complete path testing
实际结果	actual outcome	完整性	completeness
实体关系图	entity relationship diagram	伪随机	pseudo-random
实验设计	design of experiments	谓词	predicate
事务/处理	transaction	谓词数据使用	predicate data use
事务分析	transform analysis	文档测试	documentation testing
输出域	output domain	无效输入	invalid inputs
输入域	input domain	系统测试	System Testing
数据定义	data definition	系统分析	system analysis
数据定义 C-use 覆盖	data definition C-use coverage	系统集成	system integration
数据定义 C-use 使用对	data definition C-use pair	系统设计	system design
数据定义 P-use 覆盖	data definition P-use coverage	线程测试	thread testing
数据定义 P-use 使用对	data definition P-use pair	线性代码顺序和跳转	LCSAJ
数据定义使用测试	data definition-use testing	行为	behaviour
数据定义使用对	data definition-use pair	性能测试	Performance Testing
数据定义使用覆盖	data definition-use coverage	数据流测试	data flow testing
修改条件/判定测试	modified condition/ decision testing	数据流分析	data flow analysis
修改条件/判定覆盖	modified condition/ decision coverage	数据流覆盖	data flow coverage

续表

中文名	英文名	中文名	英文名
循环复杂度	cyclomatic complexity	数据流图	data flow diagram
压力测试	Stress Testing	数据确认	data validation
严格的安全性	safety critical	数据使用	data use
验证	verification	数据完整性	data integrity
一致性	consistency	数据污染	data corruption
一致性标准	conformance criterion	数据字典	data dictionary
一致性测试	Conformance Testing	死代码	dead code
一致性检查器	consistency checker	异常	anomaly
正向测试	Positive Testing	异常/例外	exception
证明	certification	因果图	cause-effect graph
指定的输入	specified input	应用软件	application software
质量保证	QA	由底向上测试	bottom-up testing
质量控制	QC	由顶向下设计	top-down design
逐步优化	stepwise refinement	语法分析	syntax testing
转换测试	conversion testing	语句	statement
桩	stub	语句测试	statement testing
状态	state	语句覆盖	statement coverage
状态图	state diagram	预置条件	precondition
状态转换	state transition	域	domain
状态转换测试	state transition testing	域测试	domain testing
桌面检查	desk checking	源代码	source code
自顶向下测试	top-down testing	源语句	source statement
自动化测试	Automated Testing	灾难恢复	disaster recovery
自动化软件质量	ASQ	真值表	truth table
走读	Walkthrough	诊断	diagnostic
组件	Component	正确性	correctness
组件测试	Component Testing		

附录 B 测试技术分类

在本书第 5 章给出了 CMU 关于测试技术方面的一个分类体系，同时还提到测试技术的分类有多种不同的标准，这里我们给出另一种测试分类方法，该方法比较实用，具体参考表 B-1。

表 B-1 测试技术分类

类型	测试技术	描述
基于人的	用户测试 (User testing)	Testing with the types of people who typically would use your product.
	Alpha 测试 (Alpha testing)	In-house testing performed by the test team.
	Beta 测试 (Beta testing)	A type of user testing that uses testers who are members of your product's target market.
	痛打臭虫 (Bug bashes)	In-house testing using secretaries, marketers and anyone who is available.
	领域专家测试 (Subject-matter expert testing)	Give the product to an expert on some issues addressed by the software and request feedback (bugs, criticisms, and compliments).
	并行测试 (Paired testing)	Two testers work together to find bugs.
	自食其果测试 (Eat your own dog-food)	Uses and relies on prerelease versions of its own software, typically waiting until the software is reliable enough for real use.
基于覆盖率的	功能测试 (Function testing)	Test every function, one by one.
	特性或功能集成测试 (Feature or function integration testing)	Test several functions together, to see how they work together.
	菜单漫游 (Menu tour)	Walk through all of the menus and dialogs in a GUI product.
	域测试 (Domain testing)	A domain is a set that includes all possible values of a variable of a function. Input domain or output domain.
	等价类分析 (Equivalence class analysis)	An equivalence class is a set of values for a variable that you consider equivalent.
	边界值测试 (Boundary testing)	If you can map them onto a number line, the boundary values are the smallest and largest members of the class.
	最佳代表测试 (Best representative testing)	A best representative of an equivalence class is a value that is at least as likely as any other value in the class to expose an error in the software.
	输入域测试分类 (Input field test catalogs or matrices)	For each type of input field, you can develop a fairly standard set of test cases and reuse it for similar fields in this product and later products.

续表

类型	测试技术	描述
基于覆盖率的	映射和测试所有编辑一个域的方法 (Map and test all the ways to edit a field)	You can often change the value of a field in several ways. For example, import data, enter data directly, copy data, copy data from script.
	逻辑从事 (Logic testing)	Logic testing attempts to check every logical relationship in the program. Cause-effect graphing is a technique for designing an extensive set of logic-based tests.
	基于状态的测试 (State-based testing)	In state-based testing, you walk the program through a large set of state transitions and check the results carefully, every time.
	路径测试 (Path testing)	Path testing involves testing many paths through the program.
	语句和分支覆盖 (Statement and branch coverage)	Testing focused on verifying every branch of the product.
	配置覆盖 (Configuration coverage)	Why do we call this a test technique? List all the types of configuration and the optimization of the effort to achieve high coverage is the test technique.
	基于规格的测试 (Specification-based testing)	Testing focused on verifying every factual claim that is made about the product in the specification.
	基于需求的测试 (Requirements-based testing)	Testing focused on proving that the program satisfies every requirement in the requirements document.
	组合测试 (Combination testing)	Testing two or more variables in combination with each other.
基于问题的	输入约束 (Input Constraints)	
	输出约束 (Output Constraints)	
	计算约束 (Computation Constraints)	
	存储或数据约束 (Storage or data Constraints)	
基于活动的	回归测试 (Regression testing)	Regression testing involves reuse of the same tests. Thress types: Bug fix regression; old bugs regression; stability regression.
	脚本测试 (Scripted testing)	Manual testing, typically done by a junior tester who follow a step-by-step procedure written by a more senior tester.
	冒烟测试 (Smoke testing)	This type of side effect regression testing is done with the goal of proving that a new build is not worth testing. Smoke tests are often automated and standardized from one build to the next.
	探险测试 (Exploratory testing)	We expect the tester to learn, throughout the proeject, about the product, its market, its risks, and the ways in which it has failed previous tests. New testers are more powerful than older because they are based on the tester's continuously increasing knowledge.
	游击测试 (Guerilla testing)	A fast and vicious attack on the program.
	场景测试 (Scenario testing)	It involves four attributes; 1. realistic; 2. complex; 3. It should be easy and quick to tell whether the program passed or failed; 4. A stakeholder.

续表

类型	测试技术	描述
基于活动的	用例流测试 (Use case flow testing)	It focusing on coverage of the important use cases. (UML)
	安装测试 (Installation testing)	Install the software in the various ways and on the various types of systems that it can be installed.
	负载测试 (Load testing)	The program or system under test is attacked, by being run on a system that is facing many demands for resources.
	长时间顺序测试 (Long sequence testing)	Testing is done overnight or for days or weeks.
	性能测试 (Performance testing)	These tests are usually run to determine how quickly the program runs.
基于评价的	自验证测试 (Self-verifying data)	The data files you use in testing carry information that lets you determine whether the output data is corrupt.
	与保存结果比较 (Comparison with saved results)	Regression testing in which pass or fail is determined by comparing the results you got today with the results from last week.
	与规格或其他权威文档比较 (Comparison with a specification or other authoritative document)	A mismatch with the specification is an error.
	启发一致性 (Heuristic consistency)	Inconsistency may be a reason to report a bug, or it may reflect intentional design variation. There are seven main consistencies.
	基于预言的测试 (Oracle-based testing)	In high-volume automated testing, an oracle is an evaluation tool that will tell you whether the program has passed or failed. The oracle is generally more trusted than the software under test.

附录 C 常见的编码错误

表 C-1 常见编码错误

类型	说明
变量使用	变量未初始化就使用。变量未初始化就使用会引起不可预知的错误，因此需要在编码的时候禁止这种使用，不管编译器是否会自动对变量进行初始化
操作使用	布尔变量的误操作。对布尔变量使用逻辑操作(&&,",!)时误写成位操作符(&^);对布尔变量使用等于操作符(==)时误写成了赋值操作符(=)
	在赋值时可能出现精度丢失。如:把浮点数赋值给一个整数,把一个无符号操作数赋值给一个有符号操作数,把一个32位整数赋值给一个16位整数等
	对有符号类型进行位移操作。位移操作一般作用于无符号操作数,如果对有符号操作数进行位移操作会出现符号位丢失现象
	对一个无符号数进行是否小于零的比较,尤其是在减1后的比较,可能会出现永真或永假结果。例如: <pre>unsigned char size; while (size-- >= 0) /* 将出现下溢 */ { ... /* program code */ }</pre>
	对浮点数直接进行相等或不相等比较(==,!=)是不合适的,应当在一定的精度范围内进行比较操作
	对字符串进行相等操作(==),例如: <pre>if (str == "abc") { }</pre> 程序员是否希望进行strcmp操作,但是实际上可能会有问题,有的编译器会把这个比较认为是指针地址的比较,而不是字符串内容的比较
	在组合条件中忽略了运算符的优先级。运算符优先级经常容易被用错,建议使用括号对,这样有利于阅读,也不容易出错
代码路径	程序中出现不可达路径。出现这种现象有可能是设计或编码的问题,也有可能是故意增加的调试代码
函数调用	函数参数需要的是无符号整数,而实际传递了一个负值进去。编译器会把负数转换成无符号整数,但这有可能不是你需要的,因此最后的结果可能是错误的
	在涉及到指针内容复制时,出现目标指针的缓冲区规模比源指针缓冲区规模小,导致目标指针溢出。这类函数包括:strcpy, memcpy, fgets等

续表

类型	说明
函数调用	<p>在条件判断中直接使用函数名，这个错误一般可能是手误引起的，然而，却很难定位，例如：</p> <pre>extern bool IsOnline(); void MyFunc { ... if(IsOnline) //程序员是要调用该函数，还是只想使用该函数指针 { ... } ... }</pre>
	函数有返回值，但是在实际使用时却没有考虑对返回值进行处理，尤其当返回值可能包含错误码的时候。这可能会成为系统的一个隐患，在某些特定的条件下造成系统的失效
	<p>函数返回了一个局部自动对象地址。例如：</p> <pre>char* Test (char* str) { char aStr[100]; return aStr; }</pre>
	函数定义的参数没有被使用到。该问题的产生有可能是为了今后的扩展使用而预留的接口，也有可能是设计或编码的时候出了问题
	没有对函数输入参数进行合法性检查。对于给其他模块使用的公用函数，要求函数内部对输入参数进行严格全面的参数检查。对于模块内的私有函数，在不是很影响效率的情况下，建议进行全面的参数检查
	在多任务操作系统环境下没有考虑函数的可重入性。在多任务的操作系统中，对于多任务共用的函数，如果该函数用到了全局变量或静态变量，需要注意到函数的可重入性
宏定义	<p>在宏定义中没有使用括号()。例如：</p> <pre>#define A B+1 Func (A*2) //这里本来希望是(B+1)*2，然而，实际上成了B+1*2</pre>
头文件使用	include 中使用了绝对路径。#include 包含的文件路径应该是相对路径，不应该使用绝对路径，经常出现错误主要是带有盘符，带有根目录符号的写法
指针使用	可能对空指针进行操作。例如，从代码的上下文空间中可以知道指针可能为空(NULL)，然而对指针的操作却没有进行判断，例如对指针使用单一操作符(*)，指针递增或递减操作符(++)或(--)等

续表

类型	说明
指针使用	<p>指针访问越界，尤其对于数组的下标操作，经常会出现这类型错误，而且容易被忽略。例如：</p> <pre>int a[10]; a[10] = 0;</pre> <p>像上面这种直接使用常量操作下标问题还容易检测出来，如果是使用变量，并且涉及到循环时，下标溢出就更具有隐藏性</p> <p>指针赋值操作中出现内存泄漏。这种情况一般在给一个被分配了内存的非空指针被赋一个新值的时候，会导致该指针原来指向的这块内存无法被释放，造成内存泄漏</p> <p>不适当的内存释放 有多种情况会造成内存释放问题：</p> <ol style="list-style-type: none">1. 多个指针指向同一个内存块，当对其中一个指针进行内存释放操作的时候，会引起其他相关指针的失效，如果这些指针还在被使用的时候，错误就会产生；2. 对一个指针进行多次释放，尤其是在指针内存被释放后，没有被赋成 NULL 的情况下；3. 对一个未经初始化的指针进行释放操作；4. 释放一个自动对象，例如：<pre>char str[1000]; ... delete[] str; ...</pre> <p>字符串越界，例如：char str[3] = "abc"；这里字符串结束符被截掉了，如果在程序中使用该字符串有可能会出现越界错误，并且这类错误很难查找</p>

附录 D 经典测试网站

表 D-1 经典测试网站

网址	简介
http://bdonline.sqe.com/	一个关于网站测试方面的网页，对这方面感兴趣的人可以参考
http://citeseer.nj.nec.com/	一个丰富的电子书库，内容很多，而且提供著作的相关文档参考和下载，是作者非常推荐的一个资料参考网站
http://groups.yahoo.com/group/LoadRunner	性能测试工具 LoadRunner 的一个论坛
http://groups.yahoo.com/group/testing-paper-announce/messages	提供网站上当前发布的软件测试资料列表
http://sate.gsfc.nasa.gov/homepage.html	软件保证中心是美国国家航天局(NASA)投资设立的一个软件可靠性和安全性研究中心，研究包括了度量、工具、风险管理等各个方面
http://seg.lit.nrc.ca/English/index.html	加拿大的一个研究软件工程质量方面的组织，可以提供研究论文的下载
http://sepo.nssc.mil	内容来自于美国 SAN DIEGO 的软件工程过程机构(Software Engineering Process Office)主页，包含软件工程知识方面的资料
http://softec.csee.usf.edu/softec/aboutsoftec.html	学校的一个软件测试中心，主要目的是为了培养企业用的测试工程师，在这里可以参考到一些概念方面的书籍资料，但不是很丰富
http://www.aptest.com/resources.html	Aptest 是一个咨询公司，在该公司的网站上有一些关于各类测试工具方面的介绍值得参考
http://www.asq.org/	是世界上最大的一个质量团体组织之一，有着比较丰富的论文资源，不过是收费的
http://www.automated-testing.com/	一个自动化软件测试和自然语言处理研究页面，属于个人网页，上面有些资源可供下载
http://www.benchmarkresources.com/	提供有关标杆测试方面的资料，也有一些其他软件测试方面的资料
http://www.betasoft.com/	包含一些流行测试工具的介绍、下载和讨论，还提供测试方面的工作介绍
http://www.brunel.ac.uk/~csstmmh2/vast/home.html	VASTT 研究组织，主要从事通过切片技术、测试技术和转换技术来验证和分析系统，对这方面技术感兴趣的人士可以在这里参考一些研究的项目及相关的一些主题信息
http://www.cc.gatech.edu/aristotle/	Aristotle 研究组织，研究软件系统分析、测试和维护等方面的技术，在测试方面的研究包括了回归测试、测试套最小化、面向对象软件测试等内容，该网站有丰富的论文资源可供下载
http://www.computer.org/	IEEE 是世界上最悠久，也是最大的计算机社会团体，它的电子图书馆拥有众多计算机方面的论文资料，是研究计算机方面的一个重要资源参考来源

续表

网址	简介
http://www.cs.colostate.edu/testing/	可靠性研究网站, 有一些可靠性方面的论文资料
http://www.cs.york.ac.uk/testsig/	约克大学的测试专业兴趣研究组网页, 有比较丰富的资料下载, 内容涵盖了测试的多个方面, 包括测试自动化、测试数据生成、面向对象软件测试、验证确认过程等
http://www.csr.ncl.ac.uk/index.html	学校里面的一个软件可靠性研究中心, 提供有关软件可靠性研究方面的一些信息和资料, 对这方面感兴趣的人可以参考
http://www.csst-technologies.com/TEST-Rx.html	主要包含有关软件测试过程的标准方法
http://www.dcs.shef.ac.uk/research/groups/vt/	学校里的一个验证和测试研究机构, 有一些相关项目和论文可供参考
http://www.esi.es/en/main/	ESI(欧洲软件组织), 提供包括 CMM 评估方面的各种服务
http://www.europeindia.org/cd02/index.htm	一个可靠性研究网站, 有可靠性方面的一些资料提供参考
http://www.fortest.org.uk/	一个测试研究网站, 研究包括了静态测试技术(如模型检查、理论证明)和动态测试(如测试自动化、特定缺陷的检查、测试有效性分析等)
http://www.grove.co.uk/	一个有关软件测试和咨询机构的网站, 有一些测试方面的课程和资料供下载
http://www.hq.nasa.gov/office/codeq/relpract/prels23.htm	NASA 可靠性设计实践资料
http://www.io.com/~wazmo/	Bret Pettichord 的主页, 他的一个热点测试页面连接非常有价值, 从中可以获得相当大的测试资料, 很有价值
http://www.iso.ch/iso/en/ISOOnline.frontpage	国际标准化组织, 提供包括 ISO 标准系统方面的各类参考资料
http://www.isse.gmu.edu/faculty/ofut/classes/821-oostest/papers.html	提供面向对象和基于构架的测试方面著作下载, 对这方面感兴趣的读者可以参考该网站, 肯定有价值
http://www.ivv.nasa.gov/	NASA 设立的独立验证和确认机构, 该机构提出了软件开发的全面验证和确认, 在此可以获得这方面的研究资料
http://www.kaner.com/	著名的测试专家 Cem Kaner 的主页, 里面有许多关于测试的专题文章, 相信对大家都有用。Cem Kaner 关于测试的最著名的书要算 <i>Testing Computer Software</i> , 这本书已成为一个测试人员的标准参考书
http://www.library.cmu.edu/Research/EngineeringAndSciences/CS+ECE/index.html	卡耐基梅隆大学网上图书馆, 在这里你可以获得有关计算机方面各类论文资料, 内容极其庞大, 是研究软件测试不可获取的资料来源之一
http://www.loadtester.com/	一个性能测试方面的网站, 提供有关性能测试、性能监控等方面的资源, 包括论文、论坛以及一些相关连接
http://www.martinig.ch/mt/index.html	关于软件工程和开发应用专业方面的时事信息和资料文件下载

续表

网址	简介
http://www.martinig.ch/mt/index.html	提供软件工程和应用开发领域的各种免费的实践知识和信息。包含了测试方面的内容
http://www.mtsu.edu/~storm/	软件测试在线资源, 包括提供目前有哪些人在研究测试, 测试工具列表连接, 测试会议, 测试新闻和讨论, 软件测试文学(包括各种测试杂志, 测试报告), 各种测试研究组织等内容
http://www.psqlconference.com/	实用软件质量技术和实用软件测试技术国际学术会议宣传网站, 每年都会举行两次
http://www.qacity.com/front.htm	测试工程师资源网站, 包含各种测试技术及相关资料下载
http://www.qaforums.com/	关于软件质量保证方面的一个论坛, 需要注册
http://www.qainusa.com/	QAI 是一个提供质量保证方面咨询的国际著名机构, 提供各种质量和测试方面证书认证
http://www.qualitytree.com/	一个测试咨询提供商, 有一些测试资料可供下载, 有几篇关于缺陷管理方面的文章值得参考
http://www.rational.com/	IBM Rational 的官方网站, 可以在这里寻找测试方面的工具信息。IBM Rational 提供测试方面一系列的工具, 比较全面
http://www.rexblackconsulting.com/Pages/publications.htm	Rex Black 的个人主页, 有一些测试和测试管理方面的资料可供下载
http://www.riceconsulting.com/	一个测试咨询提供商, 有一些测试资料可供下载, 但不多
http://www.satisfice.com/	包含 James Bach 关于软件测试和过程方面的很多论文, 尤其在启发式测试策略方面值得参考
http://www.satisfice.com/seminars.shtml	一个黑盒软件测试方面的研讨会, 主要由测试专家 Cem Kanar 和 James Bach 组织, 有一些值得下载的资料
http://www.sdmagazine.com/	软件开发杂志, 经常会有一些关于测试方面好的论文资料, 同时还包括了项目和过程改进方面的课题, 并且定期会有一些关于质量和测试方面的问题讨论
http://www.sci.cmu.edu/	著名的软件工程组织, 承担美国国防部众多软件工程项目, 在这里你可以获得各类关于工程质量和测试方面的资料。该网站提供强有力的搜索功能, 可以快速检索到你想要的论文资料, 并且可以免费下载
http://www.soft.com/Institute/HofList/	提供了网上软件质量热点连接, 包括: 专业团体组织连接、教育机构连接、商业咨询公司连接、质量相关技术会议连接、各类测试技术专题连接等
http://www.soft.com/News/QTN-Online/	质量技术时事, 提供有关测试质量方面的一些时事介绍信息, 对于关心测试和质量发展的人士来说是很很有价值的
http://www.softwareoxide.com/	包含软件工程(CMM, CMMI, 项目管理)软件测试等方面的资源
http://www.softwareqatest.com/	软件质量/测试资源中心。该中心提供了常见的有关测试方面的 FAQ 资料, 各质量/测试网站介绍, 各质量/测试工具介绍, 各质量/测试书籍介绍以及与测试相关的工作网站介绍
http://www.softwaretestinginstitute.com/	一个软件测试机构, 提供软件质量/测试方面的调查分析, 测试计划模板, 测试 WWW 的技术, 如何获得测试证书的指导, 测试方面书籍介绍, 并且提供了一个测试论坛

续表

网址	简介
http://www.sqatester.com/index.htm	一个包含各种测试和质量保证方面的技术网站, 提供咨询和培训服务, 并有一些测试人员社团组织, 特色内容是缺陷处理方面的技术
http://www.sqe.com/	一个软件质量工程服务性网站, 组织软件测试自动化、STAR-EASE、STARWEST 等方面的测试学术会议, 并提供一些相关信息资料和课程服务
http://www.stickyminds.com/	提供关于软件测试和质量保证方面的当前发展信息资料, 论文等资源
http://www.stqemagazine.com/	软件测试和质量工程杂志, 经常有一些好的论文供下载, 不过数量较少, 更多的需要通过订购获得, 内容还是很有价值的
http://www.tantara.ab.ca/	软件质量方面的一个咨询网站, 有过程改进方面的一些资料提供
http://www.tcse.org/	IEEE 的一个软件工程技术委员会, 提供技术论文下载, 并有一个功能强大的分类下载搜索功能, 可以搜索到测试类型、测试管理、测试分析等各方面资料
http://www.testing.com/	测试技术专家 Brain Marick 的主页, 包含了 Marick 研究的一些资料和论文, 该网页提供了测试模式方面的资料, 值得研究。总之, 如果对测试实践感兴趣, 该网站一定不能错过
http://www.testingcenter.com/	有一些测试方面的课程体系, 有一些价值
http://www.testingconferences.com/asiastar/home	著名的 AsiaStar 测试国际学术会议官方网站, 感兴趣的人一定不能错过
http://www.testingstuff.com/	Kerry Zallar 的个人主页, 提供一些有关培训、工具、会议、论文方面的参考信息
http://www-sqi.cit.gu.edu.au/	软件质量机构, 有一些技术资料可以供下载, 包括软件产品质量模型、再工程、软件质量改进等

附录 E 参 考 资 料

- 【1】 郑人杰. 计算机软件测试技术. 北京:清华大学出版社,1990
- 【2】 Myer G. The Art of Software Testing. Wiley,1989
- 【3】 Norm Brown. Little Book of Testing Vol I. Airlie Software Council
- 【4】 Brain Marick. Classic Testing Mistakes. Testing Foundations
- 【5】 Cem Kaner. Testing Computer Software. Van Nostrand Reinhold,1993
- 【6】 Pressman Roger. 软件工程实践者的研究方法. 北京:机械工业出版社
- 【7】 Mark Fewster, Dorothy Graham. 软件测试自动化技术. 北京:电子工业出版社
- 【8】 Lewis William. Software Testing and Continuous Quality Improvement. AUERBACH
- 【9】 Musa J, Iannino A, Okumoto K. Software Reliability Measurement Prediction Application. New York: Somepublisher, 1987
- 【10】 Howden W. Functional program testing. IEEE Transactions on Software Engineering 1980, 6:162—169
- 【11】 Thaller G. Qualitätsoptimierung der Software-Entwicklung. Das Capability Maturity Model (CMM). Braunschweig/Wiesbaden: Somepublisher, 1993
- 【12】 Crista Risley. Glass Box Testing. <http://www.cse.fau.edu/~maria/COURSES/CEN4010-SE/C13/glass.htm>
- 【13】 Thomas Raishe. BLACK BOX TESTING. <http://www.cse.fau.edu/~maria/COURSES/CEN4010-SE/C13/black.html>
- 【14】 Hausen H, Müllerburg M. Kombination von verfahren für die software-prüfung. Internationaler Kongress für Datenverarbeitung und Informationstechnologie (IKD), 1982, 111—125
- 【15】 Hausen H. Comments on practical constraints of software validation techniques. Proceedings of symposium on software validation, 1984, 323—333
- 【16】 Sneed H. Software-testen-state of the art. Software Entwicklungs-Systeme und Werkzeuge, 2 Kolloquium, 1987, 8—10
- 【17】 Hausen H, Müllerburg M, Schmidt M. Über das prüfen, messen and bewerten von software. methoden und techniken der analytischen software-qualitätssicherung. Informatik Spektrum, 1987, 10(3):123—144
- 【18】 Thaller G. Verifikation und Validation. Software Tests für Studenten und Praktiker. Wiesbaden: Somepublisher, 1994
- 【19】 Ackerman A, Fowler P, Ebenau R. Software inspection and the industrial production of software. Software Validation. Proc. Symp. Software Validation 1984, 13—40
- 【20】 Fagan M. Design and code inspection to reduce errors in program development. IBM System Journal, 1976, 15(3)
- 【21】 Howden W. Functional program testing. IEEE Transactions on Software Engineering, 1980,

6:162—169

- 【22】 Cem Kanar. Glass Box Testing. <http://www.cis.ysu.edu/~kramer/DataStructures/Intro/GlassBox.html>
- 【23】 Karat C. Cost-benefit analysis of iterative usability testing. Human Computer Interaction - INTERACT'90, Elsevier, IFIP, 1990, 351—356
- 【24】 Crellin J, Horn T, Preece J. Evaluating evaluation: A case study of the use of novel and conventional evaluation techniques in a small company. Human Computer Interaction - INTERACT'90, Elsevier, Amsterdam, 1990, 329—335
- 【25】 Lewis J, Henry S, Mack R. Integrated office software benchmarks: A case study. Human Computer Interaction - INTERACT'90, Elsevier, Amsterdam, 337—343
- 【26】 Beizer, Boris. Software Testing Techniques. 2nd Edition. New York: Van Nostrand Reinhold, 1990
- 【27】 Cem Kaner. The Impossibility of Complete Testing. Law of Software Quality Column, Software QA magazine
- 【28】 IPL. Structural Coverage Metrics. IPL, 1999
- 【29】 IPL. Advanced Coverage Metrics for OO System. IPL, 1999
- 【30】 Halstead M. Elements of Software Science. Elsevier, Amsterdam, 1977
- 【31】 VERILOG. LOGISCOPE TestChecker - Basic Concepts. VERILOG
- 【32】 Woodward M, Hedley D, Hennell M. Experience with Path Analysis and Testing of Programs. IEEE Transactions on Software Engineering, 1980, 5, VOL. SE-6, No 3, 278—286
- 【33】 Brian Marick. EXPERIENCE WITH THE COST OF DIFFERENT COVERAGE GOALS FOR TESTING. Motorola, Inc
- 【34】 Brian Marick. How to Misuse Code Coverage. Testing Foundations
- 【35】 Foster K. Error sensitive test case analysis (ESTCA). IEEE Transactions on Software Engineering, 1980, 5, 6(3): 258—264
- 【36】 Foster K. Sensitive test data for logic expressions. ACM SIGSOFT Software Engineering Notes, 1984, 4 vol. 9, no. 2, 120—126
- 【37】 Foster K. An Example of Testing Versus Verification. Software Testing, Verification & Reliability 1992, 2(1): 3—6
- 【38】 Morell J, Deimel L. Unit Analysis and Testing. SEI Curriculum Module SEI-CM-9-2.0, 1992. 6
- 【39】 IPL. Designing Unit Test Cases. IPL
- 【40】 Huang J. Program Analysis and Testing. University of Houston, 1980
- 【41】 Huang J. An Approach to Program Testing. University of Houston
- 【42】 Howden W. Applicability of Software Validation Techniques to Scientific Programs. ACM Trans. Prog. Lang. and Syst. 2, 3 (July 1980), 307—320
- 【43】 Howden W. A Functional Approach to Program Testing and Analysis. IEEE Trans. Software Eng. SE-12, 10 (Oct. 1986), 997—1005
- 【44】 Bazzichi, Franco, Ippolito Spadafora. An Auto-matic Generator for Compiler Testing. IEEE

- Trans. Software Eng. SE-8,4 (July 1982),343—353
- 【45】Duncan A, Hutchinson J. Using Attributed Grammars to Test Designs and Implementations. 5th Intl. Conf. on Software Eng. New York:IEEE, March 1981,170—178
- 【46】Weyuker E. On Testing Non-testable Programs. Computer J. 25,4 (Nov. 1982),465—470
- 【47】Gannon, John, McMullin P, Hamlet R. Data- Abstraction, Implementation, Specification, and Testing. ACM Trans. Prog. Lang. and Syst. 3,3 (July 1981),211—223
- 【48】Howden W. Methodology for the Generation of Program Test Data. IEEE Trans. Computers C-24,5 (May 1975),554—560
- 【49】Probert R. Optimal Insertion of Software Probes in Well-Delimited Programs. IEEE Trans. Software Eng. SE-8,1 (Jan. 1982),34—42
- 【50】Hecht M. Flow Analysis of Computer Programs. New York:Elsevier North-Holland,1977
- 【51】Fosdick L, Osterweil L. Data Flow Analysis in Software Reliability. ACM Computing Surveys 8,3 (Sept. 1976),305—330
- 【52】Muchnick S, Jones N. Program Flow Analysis: Theory and Applications. Englewood Cliffs, N. J. :Prentice-Hall,1981
- 【53】Osterweil L, Fosdick L. DAVE—A Validation Error Detection and Documentation System for Fortran Programs. Software-Practice and Experience 6,4 (Oct. -Dec. 1976),473—486
- 【54】Jachner, Jacek, Agarwal V. Data Flow Anomaly Detection. IEEE Trans. Software Eng. SE-10,4 (July 1984),432—437
- 【55】Weiser, Mark. Program Slicing. IEEE Trans. Software Eng. SE-10,4 (July 1984),352—357
- 【56】Korel, Bogdan, Janusz Laski. Dynamic Slicing of Computer Programs. J. Syst. and Software 13,3 (Nov. 1990),187—195
- 【57】Korel, Bogdan. The Program Dependence Graph in Static Program Testing. Information Processing Letters 24,2 (Jan. 1987),103—108
- 【58】Mills H. Software Productivity. Boston: Little, Brown, 1983
- 【59】Hamlet R. Testing Programs with Finite Sets of Data. Computer J. 20,3 (Aug. 1977),232—237
- 【60】Hamlet R. Testing Programs with the Aid of a Compiler. IEEE Trans. Software Eng. SE-3,4 (July 1977),279—290
- 【61】DeMillo R, Lipton R. Hints on Test Data Selection: Help for the Practicing Programmer. Computer 11,4 (April 1978),34—41
- 【62】DeMillo R. An Extended Overview of the Mothra Software Testing Environment. Proc. Second Workshop on Software Testing, Verification, and Analysis. Washington, D. C. : IEEE Computer Society Press, 1988, 142—151
- 【63】Morell L. A Theory of Fault-Based Testing. IEEE Trans. Software Eng. 16,9 (Aug. 1990),844—857
- 【64】Voas, Jeffrey, Morell L, Keith Miller. Predicting Where Faults Can Hide from Testing. IEEE Software 8,2 (March 1991),41—48
- 【65】Richardson D, Thompson M. The RELAY Model of Error Detection and Its Application.

- Proc. Second Workshop on Software Testing, Verification, and Analysis. Washington, D. C. : IEEE Computer Society Press, 1988, 223—230
- 【66】Morell L. Theoretical Insights into Fault-Based Testing. Proc. Second Workshop on Software Testing, Verification, and Analysis. Washington, D. C. : IEEE Computer Society Press, 1988, 45—62
- 【67】Hantler S, King J. An Introduction to Proving the Correctness of Programs. ACM Computing Surveys 8, 3 (Sept. 1976), 331—353
- 【68】Clarke L. A System to Generate Test Data and Symbolically Execute Programs. IEEE Trans. Software Eng. SE-2, 3 (Sept. 1976), 215—222
- 【69】Howden W. Symbolic Testing and the DISSECT Symbolic Evaluation System. IEEE Trans. Software Eng. SE-3, 4 (July 1977), 266—278
- 【70】Howden W. DISSECT—A Symbolic Evaluation and Program Testing System. IEEE Trans. Software Eng. SE-4, 1 (Jan. 1978), 70—73
- 【71】White L, Cohen E. A Domain Strategy for Computer Program Testing. IEEE Trans. Software Eng. SE-6, 3 (May 1980), 247—257
- 【72】Clarke L, Johnette Hassell, Richardson D. A Close Look at Domain Testing. IEEE Trans. Software Eng. SE-8, 4 (July 1982), 380—390
- 【73】Zeil S. Testing for Perturbations of Program Statements. IEEE Trans. Software Eng. SE-9, 3 (May 1983), 335—346
- 【74】Zeil S. Selectivity of Data-Flow and Control-Flow Path Criteria. Proc. Second Workshop on Software Testing, Verification, and Analysis. Washington, D. C. : IEEE Computer Society Press, 1988, 216—222
- 【75】Howden W. Weak Mutation Testing and Completeness of Test Sets. IEEE Trans. Software Eng. SE-8, 4 (July 1982), 371—379
- 【76】Howden W. Reliability of the Path Analysis Testing Strategy. IEEE Trans. Software Eng. SE-2, 3 (Sept. 1976), 208—215
- 【77】Frankl P, Weyuker E. An Applicable Family of Data Flow Testing IEEE Trans. Software Eng. 14, 10 (Oct. 1988), Criteria. 1483—1498
- 【78】Laski J, Bogdan Korel. A Data Flow Oriented Program Testing Strategy. IEEE Trans. Software Eng. SE-9, 3 (May 1983), 347—354
- 【79】Ntafos S. On Required Element Testing. IEEE Trans. Software Eng. SE-10, 6 (Nov. 1984), 795—803
- 【80】Ntafos S. A Comparison of Some Structural Testing Strategies. IEEE Trans. Software Eng. 14, 6 (June 1988), 868—874
- 【81】Podgurski, Andy, Clarke L. A Formal Model of Program Dependencies and Its Implications for Software Testing, Debugging, and Maintenance. IEEE Trans. Software Eng. 16, 9 (Sept. 1990), 965—979
- 【82】Rapps, Sandra, Weyuker E. Selecting Software Test Data Using Data Flow Information. IEEE Trans. Software Eng. SE-11, 4 (April 1985), 367—375

- 【83】 Korel, Bogdan, Janusz Laski. STAD—A System For Testing and Debugging: User Perspective. Proc. Second Workshop on Software Testing, Verification, and Analysis. Washington, D. C. ;IEEE Computer Society Press, 1988, 13—20
- 【84】 Gourlay J. A Mathematical Framework for the Investigation of Testing. IEEE Trans. Software Eng. SE-9, 6 (Nov. 1983), 686—709
- 【85】 Morell L. Theoretical Insights into Fault-Based Testing. Proc. Second Workshop on Software Testing, Verification, and Analysis. Washington, D. C. ;IEEE Computer Society Press, 1988, 45—62
- 【86】 Jefferson Offutt. The Coupling Effect; Fact or Fiction?. ACM Software Eng. Notes 14, 8 (Dec. 1989), 131—140
- 【87】 Budd T, DeMillo R, Lipton R, Sayward F. Theoretical and Empirical Studies on Using Program Mutations to Test the Functional Correctness of Programs. Conf. Record 7th Ann. ACM Symp. on Principles of Prog. Lang. New York: ACM, Jan. 1980, 220—233
- 【88】 Weyuker E, Ostrand T. Theories of Program Testing and the Application of Revealing Subdomains. IEEE Trans. Software Eng. SE-6, 3 (May 1980), 236—246
- 【89】 Howden W. Comments Analysis and Programming Errors. IEEE Trans. Software Eng. 16, 1 (Jan. 1990), 72—81
- 【90】 Howden W. Validating Programs Without Specifications. ACM Software Eng. Notes 14, 8 (Dec. 1989), 2—9
- 【91】 Hamlet R. Probable Correctness Theory. Information Processing Letters 25, 1 (April 1987), 17—25
- 【92】 Hamlet R, Ross Taylor. Partition Testing Does Not Inspire Confidence. IEEE Trans. Software Eng. 16, 12 (Dec. 1990), 1402—1411
- 【93】 DeMillo R, Lipton R. A Probabilistic Remark on Algebraic Program Testing. Information Processing Letters 7, 4 (June 1978), 193—195
- 【94】 Rowland J, Davis P. On the Use of Transcendentals for Program Testing. J. ACM 28, 1 (Jan. 1981), 181—190
- 【95】 Goodenough J, Gerhart S. Toward a Theory of Test Data Selection. IEEE Trans. Software Eng. SE-1, 2 (June 1975), 156—173
- 【96】 Jalote, Pankaj. Testing the Completeness of Specifications. IEEE Trans. Software Eng. 15, 5 (May 1989), 31—526
- 【97】 David Lee. PRINCIPLES AND METHODS OF TESTING FINITE STATE MACHINES. Mihalis Yannakakis AT&T Bell Laboratories
- 【98】 Chow T. Testing software design modeled by finite-state machines. IEEE Trans. on Software Engineering, 1978, vol. SE-4, no. 3, 178—187
- 【99】 Ghedamsi A, Bochmann G. Test result analysis and diagnostics for finite state machines. Proc. 12th Int. Conf. on Distributed Systems, 1992
- 【100】 Friedman A, Menon P. Fault Detection in Digital Circuits. Prentice-Hall, 1971
- 【101】 Cohen D, Dalal S, Fredman M, Patton G. The AETG System; a new approach to testing

- based on combinatorial design. Tech. Report TM-25261, Bell Communications Research, Morristown, N. J., 1995
- [102] Cohen D, Dalal S, Jesse Parelius, Patton. The Combinatorial Design Approach to Automatic Test Generation. in IEEE Software (Sept. 1996), 83—87
 - [103] Watson A, McCabe T. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication (Sept. 1996), 500—235
 - [104] McCabe T, Butler C. Design Complexity Measurement and Testing. Communications of the ACM, 1989. 12
 - [105] Howden W. Functional Program Testing and Analysis. McGraw-Hill, 1987
 - [106] Halstead M. Elements of Software Science. Elsevier, Amsterdam, 1977
 - [107] Allen F, Cocke J. A Program Data Flow Analysis Procedure. Communications of the ACM, vol. 19, no. 3 (March 1976), 137—147
 - [108] Hecht M, Ullman J. A Simple Algorithm for Global Data Flow Analysis Problems. SIAM J. Computing, vol. 4 (Dec. 1975), 519—532
 - [109] Kernighan B, Plaugher P. The Elements of Programming Style. McGraw-Hill, New York, N. Y., 1974
 - [110] Bieman J, Daniel Dreilinger, Lin L. Using Fault Injection to Increase Software Test Coverage. Proc. International Symposium on Software Reliability (ISSRE'96)
 - [111] Offutt A, Lee S. An empirical evaluation of weak mutation. IEEE Trans. Software Engineering (May 1994), 20 (5) : 337—345
 - [112] James Bach. James Bach on Risk-Based Testing. <http://www.stqemagazine.com/>
 - [113] Musa J. Software Reliability Engineering. McGraw-Hill, 1998
 - [114] Neil Storey. Safety-Critical Computer Systems. Addison-Wesley, 1996
 - [115] George Polya. How to Solve It?. Princeton Univ Pr, 1971
 - [116] Gause D, Weinberg G. Exploring Requirements: Quality Before Design. Dorset House, 1989
 - [117] Mearly G. A Method for Synthesizing Sequential Circuits. Bell System Technical Journal (Sept. 1955), 34 : 1045—1079
 - [118] Moor E. Gedanken- experiments on sequential machines. In Automata studies; annals of mathematical studies (34). Princeton. N. J. : Princeon University Press, 1956
 - [119] IPL. Why Bother to Unit Test?. IPL, 1996
 - [120] IPL. Organisational Approaches for Unit Testing. IPL, 1996
 - [121] Humphrey W. The Team Software ProcessSM (TSPSM). TECHNICAL REPORT CMU/SEI-2000-TR-023 ESC-TR-2000-023, 2000. 11
 - [122] Humphrey W. The Personal Software ProcessSM (PSPSM). TECHNICAL REPORT CMU/SEI-2000-TR-023 ESC-TR-2000-022, 2000. 11
 - [123] Capers Jones. Applied Software Measurement. McGraw-Hill
 - [124] IEEE. Software Verification and Validation Plan Guideline. IEEE Std 1012—1998
 - [125] IEEE. IEEE Standard for Software Unit Testing (ANSI). IEEE Std 1008—1987
 - [126] Myer G. Software Reliability: Principles and Practices. New York: John Wiley & Sons, 1976

- 【127】 Binder R. 面向对象系统的测试. 北京:人民邮电出版社,2001.4
- 【128】 Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard. Object- Oriented software engineering. Reading, Mass: Addison- Wesley, 1992
- 【129】 Bertrand Meyer. Applying Design by Contract. IEEE Computer (Oct. 1992), 25 (10): 40—51
- 【130】 Eric Gamma, Kent Beck. Test infected; Programmers love writing tests. The Java Report (July 1998), 3 (7): 37—50
- 【131】 ParaSoft. Using Design by Contract™ to Automate Java™ Software and Component Testing. <http://www.parasoft.com/jsp/products/article.jsp?articleId=1270>
- 【132】 Steve McConnell. Daily build and smoke test. IEEE Software (July 1996), 13 (4): 143—144
- 【133】 Chan W. An Overview of Integration Testing Techniques for Object- Oriented Programs. ICIS 2002, TR - 2002 - 03
- 【134】 Jean Hartmann, Claudio Imoberdorf, Michael Meisinger. UML- Based Integration Testing. ACM SIGSOFT; Volume 25, Issue 5
- 【135】 Briand L, Labiche Y, Wang Y. Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles. Proc. 12th International Symposium on Software Reliability Engineering (ISSRE' 2001), Hong Kong, 2001, 287—296, November 27—30
- 【136】 Briand L, Daly J, Wuest J. A Unified Framework for Coupling Measurement in Object- Oriented Systems. IEEE Transactions on Software Engineering, 1999, vol. 25 (1), 91—121
- 【137】 Harrold M, McGregor J, Fitzpatrick K. Incremental Testing of Object- Oriented Class Structures. Proc. 14th IEEE International Conference on Software Engineering (ICSE), Melbourne, Australia (May 1992), 68—80
- 【138】 Jorgensen P, Erickson C. Object- Oriented Integration Testing. Communications of the ACM, 1994, vol. 37 (9), 30—38
- 【139】 Kung D, Gao J, Hsia P, Toyoshima, Chen. On Regression Testing of Object- Oriented Programs. Journal of Systems Software, 1996, vol. 32 (1), 21—40
- 【140】 Chen H, Tse T, Chen T. TACCLE; a methodology for object-oriented software testing at the class and cluster levels. ACM Transactions on Software Engineering and Methodology, 2001, 10 (1): 56—109
- 【141】 Chen H, Tse T, Chan F, Chen T. In black and white; an integrated approach to class-level testing of object-oriented programs. ACM Transactions on Software Engineering and Methodology, 1998, 7 (3): 250—295
- 【142】 Humphrey W. Manager the Software Process. Addison Wesley, 1990
- 【143】 Pankaj Jalote. CMM 实践——Infosys 实施软件项目的过程 (影印版). 高等教育出版社
- 【144】 Kim Caputo. CMM 实现指南——如何改进软件过程 (影印版). 高等教育出版社
- 【145】 Ivar Jacobson. 统一软件开发过程. 北京:机械工业出版社
- 【146】 James R. Persse. CMM 实施指南. 北京:机械工业出版社
- 【147】 Booch Grady. Object- oriented design with applications. Benjamin/Cummings Publishing

Company, Inc., 1994

- 【148】 Paul Gerrard. A Unified Approach to System Functional Testing. EuroSTAR '93 (Oct. 1993), 25—28
- 【149】 James Bach. General Functionality and Stability Test Procedure. Testing Consultant Satisfice, Inc.
- 【150】 Sorkin S. System Testing Without a Specification. Coastal Technologies, 1998
- 【151】 毕军, 史美林. 计算机网络协议测试及其发展. 北京航空航天大学软件可靠性工程实验室
- 【152】 Jeffrey Voas. Testing Software for Characteristics Other than Correctness; Safety, Failure tolerance, and Security. Reliable Software Technologies (URL: <http://www.rstcorp.com>)
- 【153】 Fred Cohen. PROTECTION TESTING. <http://www.sdmagazine.com>, 1998. 9
- 【154】 Philip Koopman. Toward a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security. CSDA '98 (Nov. 1998), 11—13
- 【155】 Christopher Agruss. Software Installation Testing——How to automate tests for smooth system installation. <http://www.stqemagazine.com>, July 2000 Volume2, Issue4
- 【156】 Dave Thomas, Andy Hunt. Learning to Love Unit Testing. <http://www.stqemagazine.com>
- 【157】 Myers B, Hollan J, Cruz I. Strategic directions in human-computer interaction. ACM Computing Surveys (Dec. 1996), 28 (4) : 794—809
- 【158】 Wick D, Shehad N, Hajare A. Testing the human computer interface for the telerobotic assembly of the space station. In Proceedings of the Fifth International Conference on Human-Computer Interaction, Volume 1 of II. Special Applications, 1993, 213—218
- 【159】 Memon A, Pollack M, Soffa M. A Planning-based Approach to GUI Testing. In Proceedings of The 13th International Software/Internet Quality Week. NEW YORK: ACM Press, 2000. 5
- 【160】 Memon A, Soffa M, Pollack M. Coverage Criteria for GUI Testing. In Proceedings of The 13th International Software/Internet Quality Week. NEW YORK: ACM Press, 2000. 5
- 【161】 Gerrard P. Testing GUI Applications. EuroSTAR '97 (Nov. 1997), 24—28, Edinburgh UK
- 【162】 White L. Regression testing of GUI event interactions. In Proceedings of the International Conference on Software Maintenance (Nov. 1996), 350—358. Washington: IEEE Computer Society Press
- 【163】 Cameron Laird, Kathryn Soraiz. Testing GUI Applications. Part 1. <http://www.itworld.com/AppDev/1262/UIR010316testinggui1/>
- 【164】 Cameron Laird, Kathryn Soraiz. Testing GUI Applications. Part 2. <http://www.itworld.com/AppDev/1262/UIR010316testinggui2/>
- 【165】 阮镡. 软件可靠性与软件可靠性测试. 北京航空航天大学软件可靠性工程实验室
- 【166】 阮镡, 刘斌, 陈雪松. 软件可靠性测试及其测试环境. 北京航空航天大学软件可靠性工程实验室
- 【167】 陆民燕, 陈雪松. 软件可靠性测试及其实践. 北京航空航天大学软件可靠性工程实验室

- 【168】 Kimberly Fernsler, Philip Koopman. Robustness Testing of A Distributed Simulation Backplane. 1999 International Symposium on Software Reliability Engineering
- 【169】 Sudipto Ghosh, Mathur Aditya. TESTING FOR FAULT TOLERANCE. Software Engineering Research Center, 1398 Department of Computer Sciences, Purdue University, 1997. 9
- 【170】 DeMillo R, Li T, Mathur A. ARCHITECTURE OF TAMER; A TOOL FOR DEPENDABILITY ANALYSIS OF DISTRIBUTED FAULT-TOLERANT SYSTEMS. Department of Computer Sciences, Purdue University, 1994. 9
- 【171】 Chillarege R, Bowen N. Understanding large system failures - a fault injection experiment. In Proc. 19th Int. Symp. Fault-Tolerant Comput. , 1989, 356—363
- 【172】 Musa J. Operational profiles in reliability engineering. IEEE Software (March 1993), 14- 32
- 【173】 Goel A. Software reliability models; assumptions, limitations, and applicability. IEEE Transactions on Software Engineering(Dec. 1985), Vol. SE-11, No. 2, 1411—1423
- 【174】 Knight J. A Survey of Software Reliability Models. CS 651; Dependable Computing, Fall 2002
- 【175】 Gokhale S, Philip T, Marinos P, Trivedi K. Unification of finite-failure non-homogenous Poisson process models through test coverage. Proceedings of the 7 th IEEE International Symposium on Software Reliability Engineering (ISSRE-96), 1996. 11
- 【176】 Grottke M. Software reliability model study. PETS Project Technical Report A. 2, University of Erlangen-Nuremberg, 2001
- 【177】 Blanchard B, Fabrycky W. SYSTEMS ENGINEERING AND ANALYSIS. Third edition. 1998
- 【178】 Kenneth Crow. FAILURE MODES AND EFFECTS ANALYSIS (FMEA). DRM Associates, 2002
- 【179】 Banerjee N. Utilization of FMEA concept in software lifecycle management. Proceedings of Conference on Software Quality Management, 1995, 219—230
- 【180】 Becker J, Flick G. A practical approach to failure mode, effects and criticality analysis (FMECA) for computing systems. High-Assurance Systems Engineering Workshop, 228—236
- 【181】 Goddard P. Validating the safety of embedded real-time control systems using FMEA. 1993 Proceedings Annual Reliability and Maintainability Symposium, 227—230
- 【182】 Goddard P. Software FMEA techniques. 2000 Proceedings Annual Reliability and Maintainability Symposium, 2000, 118—123
- 【183】 Maier T. FMEA and FTA to support safe design of embedded software in safety-critical systems. Safety and Reliability of Software Based Systems. Twelfth Annual CSR Workshop, 1997, 351—367
- 【184】 NAVSO. Sneak Circuit Analysis; A Means of Verifying Design Integrity. NAVSO P-3634
- 【185】 NASA. SNEAK CIRCUIT ANALYSIS GUIDELINE FOR ELECTROMECHANICAL SYSTEMS. NASA PRACTICE NO. PD-AP-1314

- 【186】 Buratti D, Codey S. Sneak Analysis Application Guidelines. RADC-TR-82-179, Boeing Aerospace Company for Rome Air Development Center, Griffis AFB, NY 13 441, 1982. 6
- 【187】 Hill E, Bose C. Sneak Circuit Analysis of Military Systems. Boeing Aerospace Company, Seattle, WA, 2nd AIAA International Systems Safety Conference, San Diego, CA (July. 1975), 21—25, Proceedings, A77-16726-31, Newport Beach, CA, System Safety Society, 1976, 351—372
- 【188】 Miller J. Integration of Sneak Analysis with Design. RADC-TR-109, Vol. 1 of 2, Sohar Incorporated for Rome Air Development Center, Griffis AFB, NY 13441, 1990. 6
- 【189】 Walker, Frank Ellis. Sneak Circuit Analysis Automation. Boeing Aerospace, Seattle, IEEE, 1989 Proceedings Annual Reliability and Maintainability Symposium.
- 【190】 Wilson J, Clardy R. Sneak Circuit Analysis Application to Control System Design. The Boeing Company, Houston, TX, AGARD- AG- 224, In AGARD Integrity of Electronic Flight Control Systems for Aircraft Reliability, 1977. 4
- 【191】 Vogas J. Sneak Analysis of Application Specific Integrated Circuits. Boeing Aerospace Operation, Inc. , Houston, TX, AIAA- 92- 0976, 1992 Aerospace Design Conference, Irvine, CA, 1992. 2
- 【192】 陈雪松, 陆民燕, 阮镡. 基于面向对象技术的实时软件可靠性测试数据生成方法研究. 北京航空航天大学工程系统工程系 011 教研室;
- 【193】 Jelinski Z, Moranda P. Software Reliability Research. in "Statistical Computer Performance Evaluation", ed. W. Freiberger, Academic, New York, 1972, 465—485
- 【194】 Musa J, Okumoto K. A logarithmic poisson execution time model for software reliability measurement. Proceedings of the 7th international conference on Software engineering 1984. 3
- 【195】 Raihan Al-Ekram. Software Reliability Growth Modeling and Prediction. Dept. of Electrical and Computer Engineering University of Waterloo, 2002. 5
- 【196】 Katerina Goseva- Popstojanova, Trivedi K, Mathur A. How Different Architecture Based Software Reliability Models Are Related?. <http://citeseer.ist.psu.edu/goseva-popstojanova00how.html>
- 【197】 Krishnamurthy S, Mathur A. On the estimation of reliability of a software system using reliability of its components. Proceedings of the 8 th IEEE International Symposium on Software Reliability Engineering (ISSRE '97), (Nov. 1997), 146—155
- 【198】 Cheung R. A User-Oriented Software Reliability Model. IEEE Trans. Software Eng. , 1980, Vol. 6, No. 2, 118—125, 1980
- 【199】 Krishnamurthy S, Mathur A. On the Estimation of Reliability of a Software System Using Reliabilities of its Components. Proc. 8th Int ' l. Symp. Software Reliability Eng. , 1997, 146—155
- 【200】 Yacoub S, Cukic B, Ammar H. Scenario-Based Reliability Analysis of Component-Based Software. Proc. 10th Int ' l. Symp. Software Reliability Eng. , 1999, 22—31
- 【201】 Kubat P. Assessing Reliability of Modular Software. Operations Research Letters, 1989,

- Vol. 8, 35—41
- 【202】Laprie J. Dependability Evaluation of Software Systems in Operation. IEEE Trans. Software Eng., 1984, Vol. 10, No. 6, 701—714
- 【203】Rice R. User Acceptance Testing: Is It or Isn't It?. <http://www.riceconsulting.cc/uat.htm>. rcs@telepath.com
- 【204】Andrew Stricker. ALPHA and BETA TESTING General Validation Philosophy and Guidelines. <http://citl.tamu.edu,citl@tamu.edu>
- 【205】Aleritas. Beta-Testing From a Designer's Perspective. The Designers' Forum
- 【206】Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 设计模式. 北京:机械工业出版社
- 【207】Wieggers K. 软件需求. 北京:机械工业出版社
- 【208】Jeffingwell D. Calculating the Return on Investment from More Effective Requirements Management. American Programmer 10(4):13—16
- 【209】Ivar Jacobson, Grady Booch, James Rumbaugh. 统一软件开发过程. 北京:机械工业出版社
- 【210】James Rumbaugh. UML 参考手册. 北京:机械工业出版社
- 【211】斯蒂夫·迈克康奈尔. 快速软件开发——有效控制与完成进度计划. 北京:电子工业出版社
- 【212】BOHEM B. Software Engineering Economics, Prentice-Hall, 1981. FENTON, N. E. Software Metrics, Chapman & Hall, 1991
- 【213】Bench-Capon T, Castelli D, Devendeville-Brisoux L. VALIDATION, VERIFICATION AND INTEGRITY ISSUES IN EXPERT AND DATABASE SYSTEMS. Report on the 1st International Workshop. Information Research(Oct. 1998), Vol. 4 No. 2
- 【214】Eaglestone B, Ridley M. Verification, Validation and Integrity. Second Workshop on V&V and Integrity issues in Expert Systems and Databases - VVI2000
- 【215】Dymond K. A Guide to the CMM. Process Inc US
- 【216】CMU. Key Practices of the Capability Maturity Model. Version 1.1. CMU/SEI-93-TR-025, ESC-TR-93-178, 1993. 2
- 【217】Space and Naval Warfare Systems Center. SOFTWARE QUALITY ASSURANCE PROCESS. Version 1.4, 1997. 9
- 【218】Space and Naval Warfare Systems Center. SOFTWARE QUALITY ASSURANCE PLAN TEMPLATE. Version 1.2, 1997. 9
- 【219】CMU. Capability Maturity Model for Software. Version 1.1. CMU/SEI-93-TR-24, 1993. 2
- 【220】Ann Hess, Gail Becker. PEER REVIEW PROCESS. Version 1.0. 22. Software Engineering Process Office. Space and Naval Warfare Systems Center, 1998. 6
- 【221】Freedman D, Weinberg G. Handbook of walkthroughs, Inspections, and Technical Reviews, Evaluation Programs, Projects, and Products. 3rd edition. Boston: Little, Brown and Company, 1982
- 【222】Yourdon E. Structured Walkthroughs. 2nd edition. Englewood Cliffs, NJ: Prenticehall, 1979

- 【223】 IEEE. IEEE Draft Standard for Software Reviews and Audits. IEEE STD 1028-1988, IEEE Computer Society, 1988. 1
- 【224】 Fagan M. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal 15(3):182—211
- 【225】 Lawrence, Brian. Unresolved Ambiguity. American Programmer, 1996, 9(5):17—22
- 【226】 Grady R. Practical Software Metrics for Project Management and Process Improvement. Prentice-Hall ECS Professional
- 【227】 Davis A. Software Requirements: Objects, Functions, and States. Englewood Cliffs, NJ: PTR Prentice Hall, 1993
- 【228】 Norm Brown. Industrial- Strength Management Strategies, IEEE Software, 1996, 13(4): 94—103
- 【229】 Booch G, Jacobson I, Rumbaugh J. The Unified Modeling Language User Guide. Addison-Wesley, 1999, 219—241
- 【230】 Ruth Malan. Functional Requirements and Use Cases. Hewlett-Packard Company, Dana Bredemeyer, Bredemeyer Consulting
- 【231】 UML Use Case Diagrams. Engineering Notebook. C++ Report. 1998. 11
- 【232】 Cockburn, Alistair. Structuring Use Cases with Goals. Journal of Object-Oriented Programming, Sep-Oct, 1997 and Nov-Dec, 1997. Also available on <http://members.aol.com/acockburn/papers/usecases.htm>
- 【233】 Coleman, Derek. A Use Case Template; Draft for discussion. Fusion Newsletter, April 1998. Available at <http://www.hpl.hp.com/fusion/news/apr98.ppt>.
- 【234】 STSC. Requirements Engineering and Design Technology Report. 1995. 10
- 【235】 Kruchten, Philippe. A Rational Development Process. CrossTalk, 1996, 9(7):11—16
- 【236】 Dasgupta S. Technology and creativity. New York: Oxford University Press, 1996
- 【237】 Schon D. The reflective practitioner: How professionals think in action. New York: Basic Books, 1993
- 【238】 Winograd T, Bennett J, De Young L, Hartfield B. Bringing design to software. New York: Addison-Wesley Publishing Company, 1996
- 【239】 Punyashloke Mishra, Yong Zhao, Sophia Tan. UNPACKING THE BLACK-BOX OF DESIGN: FROM CONCEPT TO SOFTWARE. College of Education, Michigan State University
- 【240】 Smith G, Tabor P. The role of the artist-designer. In T. Winograd, J. Bennett, L. De Young, B. Hartfield, (Eds.), Bringing design to software. (37—57). New York: Addison-Wesley Publishing Company, 1996
- 【241】 Simon H. The sciences of the artificial. Cambridge, MA: The MIT Press, 1996
- 【242】 Rheinfrank J, Evenson S. Design languages. In T. Winograd, J. Bennett, L. De Young, B. Hartfield, (Eds.), *Bringing design to software*. (63—80). New York: Addison-Wesley Publishing Company, 1996
- 【243】 Brown J, Duguid P. Borderline issues: Social and material aspects of design. Human-Computer Interaction. 1994, 9:1, 3—36

- 【244】 Banville J. Beauty, charm, and strangeness: Science as metaphor. Science [www document]. Retrieved on January 18, 1999 from URL <http://www.sciencemag.org/cgi/content/full/281/5373/40>, 1999
- 【245】 Dijkstra E. The structure of the 'T. H. E.' multiprogramming system. CACM, 1968, vol. 18, no. 8, 453—457
- 【246】 Parnas D. On a 'buzzword': hierarchical structure. Proceedings IFIP Congress 74, 1974, 336—3390
- 【247】 Parnas D. On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM (Dec. 1972), 15(12), 1053—1058
- 【248】 Clements P, Parnas D, Weiss D. The Modular Structure of Complex Programs. IEEE Transactions on Software Engineering, Volume SE-11 (March 1985), No. 3
- 【249】 Bass L, Clements P, Kazman R. Software Architecture in Practice. Addison-Wesley, 1998
- 【250】 RICK KAZMAN. Software Architecture. Software Engineering Institute, Carnegie Mellon University
- 【251】 SEI. SEI Software Architecture. http://www.sei.cmu.edu/ata/ata_init.html.
- 【252】 David Garlan, Mary Shaw. An Introduction to Software Architecture. SEI Technical Report CMU/SEI-94-TR-21
- 【253】 IEEE. IEEE Recommended Practice for Architectural Description. Draft 3.0 of IEEE P1471, <http://www.pithecantropus.com/~awg/>, 1998. 5
- 【254】 Philippe Kruchten. The 4 + 1 view model of architecture. IEEE Software. 12(6), 1995. 11
- 【255】 Shaw M. . Larger Scale Systems Require Higher-Level Abstractions. Proceedings of Fifth International Workshop on Software Specification and Design, IEEE Computer Society, 1989, 143—146
- 【256】 Kazman R, Abowd G, Bass L, Clements P. Scenario-Based Analysis of Software Architecture. IEEE Software, 1996. 11
- 【257】 Gina Kingston. Software Design Reviews Using the Software Architecture Analysis Method: A Case Study. DSTO-RR-0170
- 【258】 Bedir Tekinerdoğ. ASAAM: Aspectual Software Architecture Analysis Method. Department of Computer Engineering, Bilkent University
- 【259】 Rick Kazman, Mark Klein, Paul Clements. ATAM: Method for Architecture Evaluation. CMU/SEI-2000-TR-004. ESC-TR-2000-004. 2000. 8
- 【260】 Abowd G, Len Bass, Paul Clements, Rick Kazman. MORALE METHODOLOGY GUIDE-BOOK Software Architecture Analysis Method (SAAM). <http://citeseer.nj.nec.com/waters99architectural.html>
- 【261】 Gregory Abowd, Len Bass, Paul Clements, Rick Kazman, Linda Northrop, Amy Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. CMU/SEI-96-TR-025. ESC-TR-96-025, 1997. 1
- 【262】 Bass L, Clements P, Kazman R, Abowd G. Analyzing Development Qualities at the Architectural Level: The Software Architecture Analysis Method. In Software Architecture in

Practice. (L. Bass, P. Clements and R. Kazman, Eds.), Addison Wesley

- 【263】 Space and Naval Warfare Systems Center San Diego. SOFTWARE TEST PLANNING AND MANAGEMENT GUIDE. VERSION 1.0, 1998. 8
- 【264】 Norm Brown. Little Book of Testing Vol II. Airlie Software Council
- 【265】 Brian Marick. Working Effectively With Developers. Testing Foundations
- 【266】 Brian Marick. The Impossibility of Complete Testing. Testing Foundations
- 【267】 Brian Marick. When Should a Test Be Automated?. Testing Foundations
- 【268】 Cem Kaner. Improving the Maintainability of Automated Test Suites. <http://www.badssoftware.com>
- 【269】 Cisco. Cisco Year 2000 Compliance Testing. http://www.cisco.com/warp/public/cc/general/year2000/prodlit/2test_ov.htm
- 【270】 Don Slutz. Massive Stochastic Testing of SQL. Microsoft Research

软件测试系列

软件测试技术概论

上海艾微软件技术有限公司 主编

古 乐 史九林 编著

清华大学出版社

北 京

软件测试系列

软件测试 技术概论

上海艾微软件技术有限公司 主编
古 乐 史九林 编著

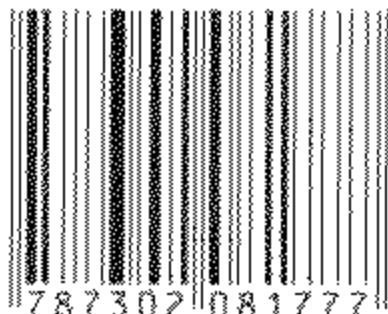


清华大学出版社

软件测试系列

- 软件测试技术概论
- 软件测试技术研究
- 软件测试自动化
- 软件测试流程
- 软件测试实战

ISBN 7-302-08177-8



9 787302 081777 >

定价: 38.00元

作 者 序

一直以来，自我感觉都非常良好，看了很多书，做了很多项目，总觉得能写出比别人更好的内容。每当有一些点子的时候，就想雄心壮志一把；然而，始终未能成书！年前，始悟“逝者如斯夫！”的道理，遂矢志著书。常感江郎才尽，言不能表意，多次停笔。踌躇，恐有误读者，深自不安。感谢妻之不断激励，不敢废，一年，始成此书。掷笔之余，窃喜大功告成。回头观之，终不能静心，浮躁之余，仓促付稿。

软件技术的发展令人不敢有丝毫之懈怠，尤其这几年，各种新兴的编程技术、开发方法、软件工具等不断涌现，使人感觉年华不在，不复当年冲浪之勇。感叹之余，深思适应之道。余喜金庸之武侠文化，所谓变化之道在于不变。只有掌握了其变化的规律，了解其不变的核心，才能适应现今社会的发展。至于软件之道，不论其语言如何之变化，工具如何之花巧，这对于一个项目的成败来说并不是最重要的，关键在于其分析的思想，设计的核心，过程的监控。不管技术如何之变化，这个思想是不变的，并且只有当你掌握了这个思想，你的经验和价值才会随着年龄的增长而增长，而不是像普通的程序员一样，过了某个年龄段之后就发“廉颇老矣，尚能饭否”的感叹。变与不变是辩证的，只有在变化的事物中，把握其不变的主题，才能立于不败之地。

本书的重点在于介绍软件测试的一些基本概念和方法，涵盖了业界出现的大部分测试领域内的知识，是一本比较全面的测试方法介绍书籍。

本书具有以下特色：

1. 内容涉及面广。综合考虑了测试的方方面面，内容涵盖了动态测试到静态测试，白盒测试到黑盒测试，单元测试到系统测试等多个测试领域。对于近几年才出现的一些测试方法也进行了阐述和比较。

2. 理论与实践结合。本书不是一本纯理论方面的书籍。书中很多涉及理论的内容多是通过实践经验的方式来阐述的。因此，可能理论的严谨性会差一些，但是却更容易被理解和接受。同时，作者在很多地方都对一些非常有用的测试经验进行了总结。尤其在最后一章，给出了测试的一些重要原则和实践方法。

3. 强调测试分析和测试设计的重要性。本书认为测试的质量关键靠测试分析和测试设计的质量。因此，在本书中无论是讲单元测试、集成测试还是系统测试，都讲究如何进行测试数据分析，测试用例设计。

4. 引入过程概念，强调全面质量管理。产品的质量不是突然获得的，而是靠过程来保证的。因此，对于一个产品的测试，如果要保证测试的质量，也是需要一个过程的。软件工程研究指出：做一个产品应当遵循一个规范的工程过程。同样，做测试也要遵循一个定义的过程。在本书中，强调全面的质量管理。从最初的需求测试到最后的系统测试、验收测试，都需要遵循一定的过程。只有保证了每个阶段的过程质量，产品最终的质量才有可能得到保证。那种期望最后能够突然获得质量的想法是不现实的。

5. 提出需求测试和设计测试的思想。测试的一个重要原则是尽早测试。然而，如何

对需求进行测试和对设计进行测试是目前国内外同行研究的一个课题。本书对这两个领域进行了比较全面的描述,一方面吸收国外一些好的想法。同时根据作者的实践经验提出了自己的见解(说明:书中上角标注的【1】【2】等指本段话引自附录 E 中相应的参考文献)。

如何学习此书? 作者认为软件测试是从软件工程中演化出来的一门学科,并且还在不断发展当中。本书虽介绍了业界很多的测试概念和方法;然而,这只是沧海之一粟。我们不能穷尽测试的任何方方面面,也不需要这样做。就像前面所说的,你需要把握住测试的核心。学习任何一项测试技术,不仅要知其然,还要知其所以然。从抽象到实践,再从实践回归。只有这样,才能更好地领悟书中所涉及的概念和方法。

软件测试作为一门软件工程方面的学科,要求具有一定的程序设计经验和分析设计经验。掌握一门编程语言,如:C 语言,并且了解一定的软件开发过程知识对于本书的理解是十分有益和必要的。对于有项目实际开发经验的人员来说,学习本书将会更容易一些。

在本书写作过程中得到多方面的帮助、指导和支持。首先感谢出版社给了我出版的机会,否则我就只能孤芳自赏了。其次要感谢我最深爱着的妻子,要不是受她高涨的工作热情的影响,并给我不断的鼓励,本书也就不会完成了。最后要感谢上海艾微软件技术有限公司的同事们,大家为着共同的理想走到了一起,彼此鼓励、相互支持,相信总有一天会获得成功。

鉴于作者才疏学浅,书中不乏疏漏和错误,欢迎读者批评指正。

编者于上海

2003 年 6 月 22 日

kulerj@hotmail.com

上海艾微软件技术有限公司

support@ivei.com

<http://www.ivei.com>

内 容 简 介

本书是一本比较全面地介绍软件测试方法的书籍,先介绍测试技术的发展历史和现状;然后,作为测试的一个基础,介绍了白盒测试、黑盒测试以及测试覆盖率等几个重要概念,并充分分析了业界在这几个概念方面的研究成果;之后从全流程测试的角度按动态测试和静态测试两个方面,分别介绍了单元测试、集成测试、系统测试、验证和确认过程、需求测试和设计测试等内容。作为静态测试的一个重要手段,本书还对同行评审的概念和方法进行了阐述。最后,作者总结了测试的基本原则和一些好的实践经验。

本书可以作为大学本科高年级学生或研究生教材,也可以作为本科低年级学生学习的参考书。对于软件工程师和测试工程师来说,本书是一本很好的指导书。

版权所有,翻印必究。

本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

软件测试技术概论/古乐,史九林编著. —北京:清华大学出版社,2004.4

ISBN 7-302-08177-8

I. 软… II. ①古… ②史… III. 软件—测试 IV. TP311.5

中国版本图书馆CIP数据核字(2004)第014383号

出 版 者:清华大学出版社

<http://www.tup.com.cn>

社总机:010-62770175

地 址:北京清华大学学研大厦

邮 编:100084

客户服务:010-62776969

组稿编辑:丁 岭

文稿编辑:闫红梅

封面设计:付剑飞

印 刷 者:北京鑫丰华彩印有限公司

装 订 者:三河市李旗庄少明装订厂

发 行 者:新华书店总店北京发行所

开 本:185×260 印张:29.25 字数:726千字

版 次:2004年4月第1版 2004年4月第1次印刷

书 号:ISBN 7-302-08177-8/TP·5905

印 数:1~4000

定 价:38.00元

本书如存在文字不清、漏印以及缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。联系电话:(010)62770175-3103 或(010)62795704